SONY

SONY
COMPUTER
ENTERTAINMENT ®

Cell Broadband Engine Architecture

Version 1.0

August 8, 2005

All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change Sony and SCEI product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of Sony and SCEI or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments can vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will Sony and SCEI be liable for damages arising directly or indirectly from any use of the information contained in this document.

Sony Corporation
6-7-35 Kitashinagawa
Shinagawa-ku, Tokyo, 141-0001 Japan

Sony Computer Entertainment Inc.
2-6-21 Minami-Aoyama, Minato-ku,
Tokyo, 107-0062 Japan

The Sony home page can be found at **http://www.sony.net**
The SCEI home page can be found at **http://www.scei.co.jp**

The Cell Broadband Engine home page can be found at **http://cell.scei.co.jp**

August 8, 2005
Version 1.0

Contents

List of Figures

List of Tables

Preface

This document defines the Cell Broadband Engine Architecture (CBEA). The information contained in this document allows various CBEA-compliant processor implementations to meet the needs of a wide variety of systems and applications. Compatibility with the CBEA allows applications and privileged software to migrate from one implementation to another with minor changes.

For a specific implementation of the CBEA, see the specific implementation documentation.

Who Should Read This Manual

This manual is intended for designers who plan to develop products using the CBEA.Readers of this manual should be familiar with the documents listed in *Related Publications* on page 16. In addition, individual implementations of the CBEA should have their own documentation relating to that implementation.

Document Organization

The CBEA document is divided into two environments; the User Mode Environment (UME) and the Privileged Mode Environment (PME). In general, the UME describes the commands and facilities available to an application while the PME describes the facilities available to an operating system or hypervisor code. Together, these two environments define the CBEA. Implementation details and compliance with the architecture for a specific implementation are provided separately.

Document Division	Description
Front Matter and Introduction	The sections of this part include: • *Title Page* — Document classification, version number, release date, copyright, and disclaimer information. • *Contents* • *List of Figures* • *List of Tables* • *Preface* — Describes this document, related documents, user requirements, and other general information. • *Introduction to Cell Broadband Engine Architecture* Provides a high-level overview of the Cell Broadband Engine Architecture (CBEA).
User Mode Environment	Defines the base instruction set, command set, storage models, and facilities available to an application programmer, as well as compatibility with the PowerPC Architecture. The sections of this part include: • Overview • Storage Models • PowerPC Processor Element • Synergistic Processor Unit • Memory Flow Controller • MFC Commands • Problem State Memory-Mapped Registers • Synergistic Processor Unit Channels • Storage Access Ordering • SPU Isolation Facility.

Document Division	Description
Privileged Mode Environment	Describes the additional instructions and facilities, beyond those defined in User Mode Environment that are provided by the CBEA This division covers instructions and facilities not available to the application programmer, which affect storage control, interrupts, and timing facilities. The sections of this part include: • Overview • Compatibility with PowerPC Architecture, Book III • Storage Addressing • Real-Mode Storage Control Facilities (Optional) • MFC Privileged Facilities • SPU Privileged Facilities • SPE Context Save and Restore • PPE Address Range Facility • Cache Replacement Management Facility • Resource Allocation Management • Interrupt Facilities • Power Management • Version Control.
Back Matter and Revision Notices	The sections of this part include: • *Appendix A Memory Maps* • *Appendix B SPU Channel Map* • *Appendix C CBEA-Specific PPE Special Purpose Registers* • *Appendix D Defined Commands* • *Appendix E Extensions to the PowerPC Architecture* • *Appendix F Examples of Access Ordering* • *Glossary* • *Index* • *Revision Log*

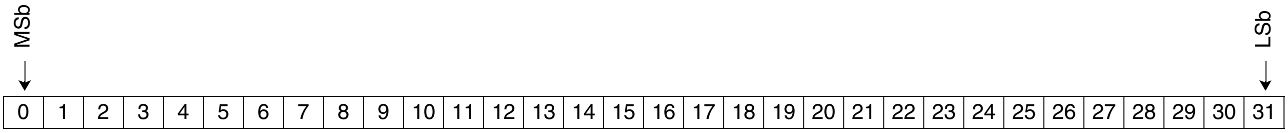
Related Publications

A list of materials related to the CBEA follows.

Title	Version	Date
PowerPC User Instruction Set Architecture, Book I	2.02	January 28, 2005
PowerPC Virtual Environment Architecture, Book II	2.02	January 28, 2005
PowerPC Operating Environment Architecture, Book III	2.02	January 28, 2005
PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors (G522-0290-01)	1.0	February 21, 2000
Synergistic Processor Unit Instruction Set Architecture	1.0	August 2005

Conventions and Notation

In this document, standard IBM notation is used, meaning that bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most significant bit and bit 31 is the least-significant bit.



Notation for bit encoding is as follows:

- Hexadecimal values are preceded by x and enclosed in single quotation marks. For example: x‘0A00’.
- Binary values in sentences appear in single quotation marks. For example: ‘1010’.

The following software documentation conventions are used in this manual:

1. Command (or instruction) names are written in **bold** type. For example: **put**, **get, rdch, wrch, rchcnt...**
2. Variables are written in italic type. Required parameters are enclosed in angle brackets. Optional parameters are enclosed in brackets. For example: **get*<f,b>[s]***.

In some cases, registers are referred to by the register mnemonic. Fields are then referred to by the register mnemonic followed by the field name enclosed in brackets. An equal sign followed by a value indicates the value to which the field is set. For example, MFC_SR1[R] = 0). For more information, see *Referencing Registers, Fields, and Bit Ranges*.

The following symbols are used in this document:

&	bitwise AND
|	bitwise OR
%	modulus
=	equal to
! =	not equal to
x ≥	greater than or equal to
x ≤	less than or equal to
x >> y	shift to the right; for example, 6 >> 2 = 1; least significant y-bits are dropped
x << y	shift to the left; for example, 3 << 2 = 12; least significant y-bits are replaced zeros

Referencing Registers, Fields, and Bit Ranges

Registers are referenced by their full name or by their short name (also called the register mnemonic). Fields within registers are referenced by their full field name or by their field name. The field name or names are enclosed in brackets [ ].The following table describes how registers, fields, and bit ranges are referenced in this document and provides examples of the references.

Type of Reference	Format	Example
Reference to a specific register and a specific field using the register short name and the field name(s), bit number(s), or bit range.	Register_Short_Name[Bit_FieldName]	MSR[FE0]
	Register_Short_Name[Bit_Number]	MSR[52]
	Register_Short_Name[Field_Name1, Field_Name2]	MSR[FE0, FE1]
	Register_Short_Name[Bit_Number, Bit_Number]	MSR[52, 55]
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number]	MSR[39:44]
Reference to a specific register and the setting for a specific field, bit, or range of bits using the register short name and the field name(s), bit number(s), or bit range that is followed by an equal sign (=) and a value that field.	Register_Short_Name[Field_Name] = ‘*n*’ (where *n* is a binary value for the bit or bit range)	MSR[FE0] =’1’
	Register_Short_Name[Field_Name] =x‘*n*’ (where *n* is a hexadecimal value for the bit or bit range)	MSR[FE] = x‘1’
	Register_Short_Name[Bit_Number] = ‘*n*’ (where *n* is a binary value for the bit or bit range)	MSR[52] = ‘0’
	Register_Short_Name[Bit_Number]] = x‘*n*’ (where *n* is a hexadecimal value for the bit or bit range)	MSR[52] = x‘0’
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number] = ‘*n*’ (where *n* is the binary or hex value for the bit or bit range)	MSR[39:43] = ‘10010’ [39:43] = ‘10010’
	Register_Short_Name[Starting_Bit_Number:Ending_Bit_Number] = ‘*n*’ (where *n* is the hexadecimal value for the field or bits)	MSR[39:43] = x‘11’ [39:43] = x‘11’

Note: The register short name is also called the register mnemonic.

Endian Order

The PowerPC Architecture supports both big-endian and little-endian byte-ordering modes. A complete description of these modes is given in *Book I* of the *PowerPC Architecture* document. For more information about big-endian and little-endian byte ordering, see the following sections in that book:

- *Big-Endian Mapping*
- *Little-Endian Mapping*

The CBEA supports only big-endian byte ordering. Therefore, PPEs in a CBEA-compliant implementation are *not* required to support the little-endian byte-ordering mode as defined in the PowerPC Architecture. Synergistic processor units (SPUs) do *not* implement the optional little-endian byte-ordering mode.

The PPE structure mapping is identical to that used for the SPUs in a CBEA-compliant system. Since the CBEA supports only big-endian byte ordering, the memory flow controller (MFC) direct memory access (DMA) command and control registers do *not* implement the optional little-endian byte-ordering mode. The DMA data transfers themselves are simply byte moves, without regard to the numerical significance of any byte. Thus, the big-endian or little-endian issue becomes irrelevant to the actual movement of a block of data. The byte-order mapping only becomes significant when data is fetched or interpreted, for example by a processor, or by an MFC.

1. Introduction to Cell Broadband Engine Architecture

The Cell Broadband Engine Architecture (CBEA) defines a processor structure directed toward distributed processing. The intent of the architecture is to allow implementation of a wide range of single or multiple processor and memory configurations, in order to optimally address many different systems and application requirements.

The CBEA is divided into two environments: the User Mode Environment (UME) and the Privileged Mode Environment (PME). The UME describes the commands and facilities available to an application programmer while the PME should only be used by software in privileged mode, such as operating systems.

This document does not cover all aspects of the CBEA. *PowerPC Architecture, Books I - III* define the PowerPC Processor Element (PPE) components. The Synergistic Processor Unit Instruction Set Architecture document defines the Synergistic Processor Unit (SPU) components. The focus of this document is on the infrastructure around the computational, data movement, communication, synchronization, and resource management components. Together, these three documents are required for a complete definition of the CBEA.

Since the PowerPC Architecture™ is a standalone processor architecture, this document outlines the necessary optional features and extensions required by the CBEA. The *Synergistic Processor Unit Instruction Set Architecture* focuses on the instruction set for the SPU component. The SPU is detailed in a separate document to allow for its instruction set to be used in non-CBEA-compliant implementations and in other architectures.

1.1 Broadband Processor Organization

Physically, a CBEA-compliant processor can consist of a single chip, a multi-chip module (or modules), or multiple single-chip modules on a motherboard or other second-level package, depending on the technology used and the cost/performance characteristics of the intended design point.

Logically, the CBEA defines four separate types of functional components: the PowerPC Processor Element (PPE), the Synergistic Processor Unit (SPU), the Memory Flow Controller (MFC) and the Internal Interrupt Controller (IIC). The computational units in the CBEA-compliant processor are the PPE and the SPU. Each SPU must have a dedicated local storage, a dedicated MFC with its associated Memory Management Unit (MMU), and Replacement Management Table (RMT). The combination of these components is referred to as an SPU Element, or SPE.

Figure 1-1 on page 20 illustrates a CBEA-compliant processor in which a group of SPEs share a single SL1 cache (an SL1 cache is a first-level cache for DMA transfers between local storage and main storage) and a group of PPEs share a single second-level (L2) cache. (While caches are shown for the SPE and PPE, they are considered optional in the CBEA.) Also included in the illustration are two controllers typically found in a processor: a Memory Interface Controller (MIC) and a Bus Interface Controller (BIC). Connecting the various units within the processor is a Element Interconnect Bus (EIB). Since the requirements for the MIC, BIC, and EIB vary widely between implementations, the definition for these units are beyond the scope of the CBEA.

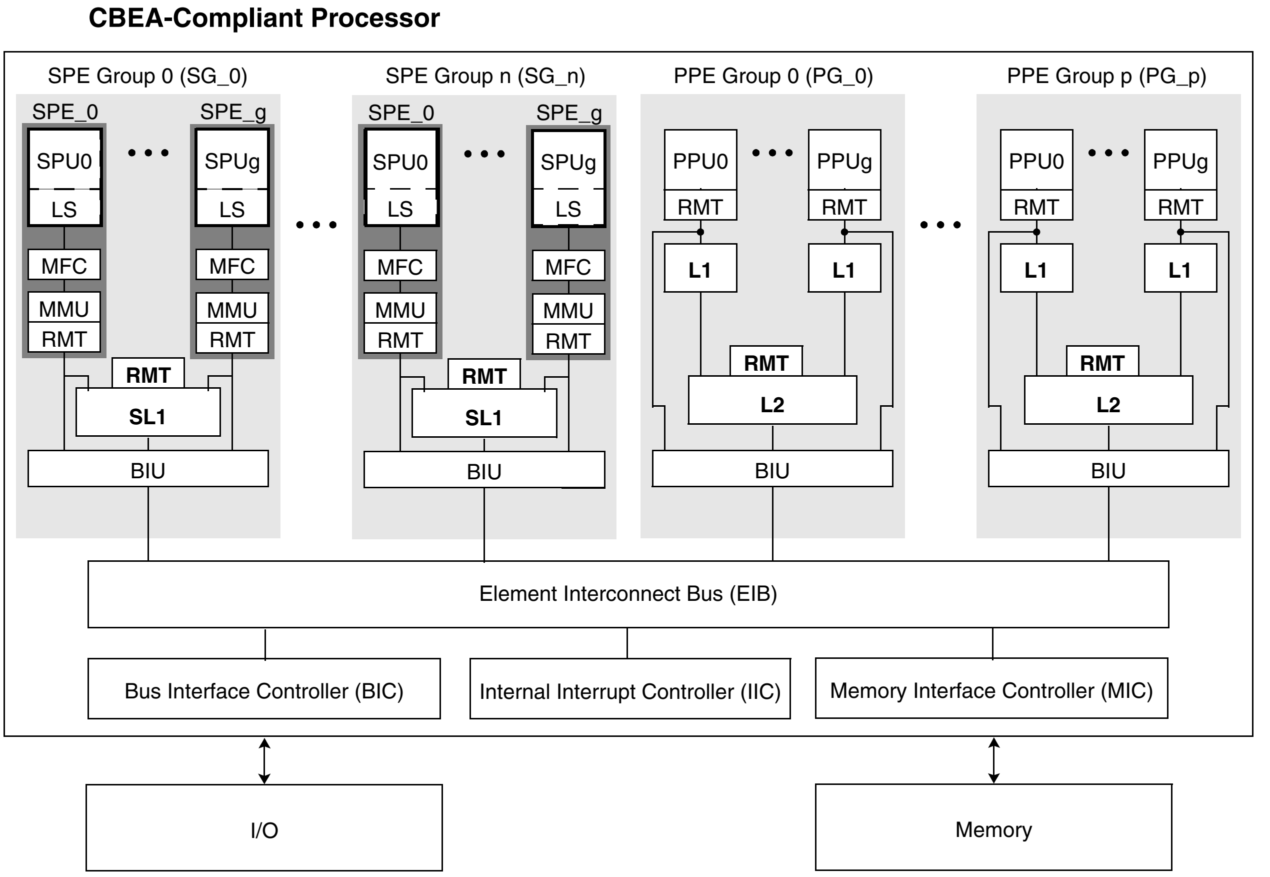
A processor can include multiple groups of PPEs (PPE groups) and multiple groups of SPEs (SPE groups). Hardware resources can be shared between units within a group. However, the SPEs and PPEs must appear to software as independent elements.

Each SPU in an SPE group has its own local storage area and a dedicated MFC that includes an associated memory management unit (MMU), which can hold and process memory-protection and access-permission information.

In summary, a CBEA-compliant system must include the following:

- One or more PowerPC Processor Elements (PPEs)
- One or more Synergistic Processor Elements (SPEs), which are the combination of a Synergistic Processor Units (SPUs), a local storage area, and a Memory Flow Controller (MFC)
- One Internal Interrupt Controller (IIC)
- One Element Interconnect Bus (EIB) for connecting units within the processor

Figure 1-1. CBEA-Compliant Processor System



BIC	Bus interface controller
BIU	Bus interface unit
IIC	Internal interrupt controller
L1	Memory cache internal to the CPU
L2	Memory cache external to the CPU
LS	Local storage
MFC	Memory flow controller
MIC	Memory interface controller
MMU	Memory management unit
PPE	PowerPC processor element
PPU	PowerPC processor unit
RMT	Replacement-management table
SL1	First-level cache
SPE	Synergistic processor element
SPU	Synergistic processor unit

1.1.1 PowerPC Processor Element

A CBEA-compliant processor includes one or more PPEs. These are 64-bit PowerPC Processor Units (PPUs) with associated caches that conform to *PowerPC Architecture, Books I - III.* For more detail on compatibility issues, refer to *Section 4 PowerPC Processor Element* beginning on page 39.

A CBEA-compliant system must include a vector multimedia extension unit in the PPE.

The PPEs are general-purpose processing units, which can access system management resources (such as the memory-protection tables, for example). Hardware resources defined in the CBEA are mapped explicitly to the real address space as seen by the PPEs. Therefore, any PPE can address any of these resources directly by using an appropriate effective address value.

A primary function of the PPEs is the management and allocation of tasks for the SPEs in a system.

1.1.2 Synergistic Processor Unit

A CBEA-compliant processor includes one or more SPUs. The SPUs are less complex computational units than PPEs, in that they do not perform any system management functions. They have a single instruction, multiple data (SIMD) capability and typically process data and initiate any required data transfers (subject to access properties set up by a PPE) in order to perform their allocated tasks.

The purpose of the SPU is to enable applications that require a higher computational unit density and can effectively use the provided instruction set. A significant number of SPUs in a system, managed by the PPEs, allow for cost-effective processing over a wide range of applications.

The SPUs implement a new instruction set architecture. The main characteristics of this architecture are described in *Section 5 Synergistic Processor Unit* beginning on page 41.

1.1.3 Memory Flow Controller

MFC components are essentially the data transfer engines. They provide the primary method for data transfer, protection, and synchronization between main storage and the local storage. An MFC command describes the transfer to be performed. A principal architectural objective of the MFC is to perform these data transfer operations in as fast and as fair a manner as possible, thereby maximizing the overall throughput of a CBEA-compliant processor.

Commands that transfer data are referred to as MFC DMA commands. These commands are converted into DMA transfers between the local storage domain and main storage domain. Each MFC can typically support multiple DMA transfers at the same time and can maintain and process multiple MFC commands.

In order to accomplish this, the MFC maintains and processes queues of MFC commands. Each MFC provides one queue for the associated SPU (MFC SPU command queue) and one queue for other processors and devices (MFC proxy command queue). Logically, a set of MFC queues is always associated with each SPU in a CBEA-compliant processor, but some implementations of the architecture can share a single physical MFC between multiple SPUs, such as an SPU group. In such cases, all the MFC facilities must appear to software as independent for each SPU.

Each MFC DMA data transfer command request involves both a local storage address (LSA) and an effective address (EA). The local storage address can directly address only the local storage area of its associated SPU.

The effective address has a more general application, in that it can reference main storage, including all the SPU local storage areas, if they are aliased into the real address space (that is, if MFC_SR1[D] is set to ‘1’). (See *Section 15.1 MFC State Register One* beginning on page 197 for more information.)

The MMU in an MFC supports the storage addressing model described in *PowerPC Architecture, Book III* and the extensions described in *Section 14 Storage Addressing* beginning on page 177 of this document.

An MFC presents two types of interfaces: one to the SPUs and another to all other processors and devices in a processing group.

- **SPU Channel:** The SPUs use a channel interface to control the MFC. In this case, code running on an SPU can only access the MFC SPU command queue for that SPU.
- **Memory-Mapped Register:** Other processors and devices control the MFC by using memory-mapped registers. It is possible for any processor and device in the system to control an MFC and to issue MFC proxy command requests on behalf of the SPU.

The MFC also supports bandwidth reservation and data synchronization features.

1.1.4 Internal Interrupt Controller Component

The IIC component manages the priority of the interrupts presented to the PPEs. The main purpose of the IIC is to allow interrupts from the other components in the processor to be handled without using the main system interrupt controller. The IIC is really a second level controller. It is intended to handle all interrupts internal to a CBEA-compliant processor or within a multiprocessor system of CBEA-compliant processors. The system interrupt controller will typically handle all interrupts external to the CBEA-compliant processor.

In a CBEA-compliant system, software must first check the IIC to determine if the interrupt was sourced from an external system interrupt controller. The IIC is not intended to replace the main system interrupt controller for handling interrupts from all I/O devices.

1.1.5 Storage Types

There are two types of storage domains within the CBEA: local storage domain and main storage domain. The local storage of the Synergistic Processor Elements (SPEs) exists in the local storage domain. All other facilities and memory are in the main storage domain.

Local storage consists of one or more separate areas of memory storage, each one associated with a specific SPU. Each SPU can only execute instructions (including data load and data store operations) from within its own associated local storage domain. Therefore, any required data transfers to, or from, storage elsewhere in a system must always be performed by issuing an MFC DMA command to transfer data between the local storage domain (of the individual SPU) and the main storage domain, unless local storage aliasing is enabled.

1.1.5.1 Local Storage Addressing

An SPU program references its local storage domain using a local address. However, privileged software can allow the local storage domain of the SPU to be aliased into main storage domain by setting the D bit of the MFC_SR1 to ‘1’. Each local storage area is assigned a real address within the main storage domain. (A real address is either the address of a byte in the system memory, or a byte on an I/O device.) This allows privileged software to map a local storage area into the effective address space of an application to allow DMA transfers between the local storage of one SPU and the local storage of another SPU.

Other processors or devices with access to the main storage domain can directly access the local storage area, which has been aliased into the main storage domain using the effective address or I/O bus address that has been mapped through a translation method to the real address space represented by the main storage domain.

Data transfers that use the local storage area aliased in the main storage domain should do so as caching inhibited, since these accesses are not coherent with the SPU local storage accesses (that is, SPU load, store, instruction fetch) in its local storage domain. Aliasing the local storage areas into the real address space of the main storage domain allows any other processors or devices, which have access to the main storage area, direct access to local storage. However, since aliased local storage must be treated as non-cacheable, transferring a large amount of data using the PPE load and store instructions can result in poor performance. Data transfers between the local storage domain and the main storage domain should use the MFC DMA commands to avoid stalls.

1.1.5.2 Main Storage Addressing

The addressing of main storage in the CBEA is compatible with the addressing defined in the PowerPC Architecture. The CBEA builds upon the concepts of the PowerPC Architecture and extends them to addressing of main storage by the MFCs.

An application program executing in an SPU or in any other processor or device uses an effective address to access main storage. The effective address is computed when the PPE performs a load, store, branch, or cache instruction, and when it fetches the next sequential instruction. An SPU program must provide the effective address as a parameter in an MFC command. The effective address is translated to a real address according to the procedures described in the overview of address translation in *PowerPC Architecture, Book III*. The real address is the location in main storage which is referenced by the translated effective address.

Main storage is shared by all PPEs, MFCs, and I/O devices in a system. All information held in this level of storage is visible to all processors and to all devices in the system. This storage area can either be uniform in structure, or can be part of a hierarchical cache structure. Programs reference this level of storage using an effective address.

1.1.5.3 Main Storage Attributes

The main storage of a system typically includes both general-purpose and nonvolatile storage, as well as special-purpose hardware registers or arrays used for functions such as system configuration, data-transfer synchronization, memory-mapped I/O, and I/O subsystems.

Table 1-1 lists the sizes of address spaces in main storage.

Table 1-1. Sizes of Main Storage Address Spaces

Address Space	Size	Description
Real Address Space	2^m bytes	where $m \leq 62$
Effective Address Space	2^{64} bytes	An effective address is translated to a virtual address using the segment lookaside buffer (SLB).
Virtual Address Space	2^n bytes	where $65 \leq n \leq 80$ A virtual address is translated to a real address using the page table.
Note: The values of “m,” “n,” and “p” are implementation-dependent.		

Table 1-1. Sizes of Main Storage Address Spaces

Address Space	Size	Description
Real Page (Base)	2^{12} bytes	
Virtual Page	2^{p} bytes	where $12 \le p \le 28$ Up to eight page sizes can be supported simultaneously. A small 4-KB (p = 12) page is always supported. The number of large pages and their sizes are implementation-dependent.
Segment	2^{28} bytes	The number of virtual segments is $2^{(n - 28)}$ where $65 \le n \le 80$
Note: The values of “m,” “n,” and “p” are implementation-dependent.		

1.2 Cache Replacement Management Facility

The CBEA includes an optional facility for managing critical resources within the processor and system. The resources targeted for management under the CBEA are the translation lookaside buffers (TLBs) and data and instruction caches. Management of these resources is controlled by implementation-dependent tables. Tables for managing TLBs and caches are referred to as replacement management tables (RMTs). These tables are optional under the CBEA, but it is strongly recommended that an implementation provide a table for each critical resource, which can be a bottleneck in the system.

An SPE group can also contain an optional cache hierarchy, the SL1 caches, which represent first level caches for DMA transfers. The SL1 caches can also contain an optional RMTs. The features and operation of the SL1 caches are described in *Section 7.7 Storage Control Commands* beginning on page 57.
The Replacement Management Tables (RMT) are described in *Section 19 Cache Replacement Management Facility* beginning on page 231.

1.3 Instructions, Commands, and Facilities

Instructions define an operation to be performed by a processing element. Descriptions of the supported instructions for the PPE are given in the PowerPC Architecture. In the CBEA, *Section 4 PowerPC Processor Element* beginning on page 39, and *Section 13 Compatibility with PowerPC Architecture, Book III* beginning on page 175 describe the features and extensions that are optional in the PowerPC Architecture, but that are required by the CBEA. *Section 5 Synergistic Processor Unit* beginning on page 41 defines the version of the Synergistic Processor Instruction Set Architecture supported by the CBEA.

Commands define operations to be performed by the MFC. Descriptions of the supported commands are given in *Section 7 MFC Commands* beginning on page 47.

Facilities is a term used to describe functionality accessed through the main storage domain for processors or devices having such access, or by SPUs through the use of channel instructions. In some cases, a facility can be accessed through a single main storage address or by a single SPU channel. A list of the facilities supported by the CBEA is shown in *Section 6.1 MFC Facilities* beginning on page 44.

1.4 Reserved Fields and Registers

All reserved fields should be zero. MFC commands with reserved fields not set to zero are considered invalid. For a description of the invalid instructions and invalid MFC forms, see *Section 7 MFC Commands* beginning on page 47.

The handling of reserved bits in a specific instantiation of the CBEA is implementation-dependent. For each reserved bit, an implementation will either:

- Ignore the reserved bits on writes and return zeros for the reserved bits on reads; or
- Maintain the state of the reserved bits.

All reserved registers should be set to zero, unless otherwise indicated. All reserved registers must be decoded by an implementation for both reads and writes. The handling of reserved register values in a specific instantiation of the CBEA is implementation-dependent. For each reserved register, an implementation will either:

- Ignore the value on writes and return zeros for reads of the register, or
- Maintain the state of the reserved register.

Fields and registers which are currently defined as reserved can become defined in future versions of the CBEA.

Programming Note:

Software must preserve the state of the reserved bits. In order to accomplish this in an implementation-independent fashion, software should:

- Initialize all reserved bits to zero.
- Alter only the defined bits by reading the register, modifying the desired bits, and writing the new value back to the register.

In some cases, it should be possible for software to always set the reserved bits to zero. Refer to the register definition for the proper method of handling reserved bits.

Software should not use reserved bits or reserved registers to maintain the software state for any purpose.

Implementation Note:

An implementation of the CBEA should never use reserved fields or registers for an implementation-dependent purpose.

All defined, reserved, and implementation-dependent registers must be decoded for both reads and writes. Furthermore, the range of address for a given area in the memory map must be a multiple of at least 4K bytes (that is, the smallest page addresses). An implementation should consider the range to be a multiple of a larger supported page size to minimize the number of page table entries required to map the address range. See *Appendix A Memory Maps* on page 269 for more details.

1.5 Implementation-Dependent Fields and Registers

Implementation-dependent fields and registers are provided in the CBEA. These fields and registers are intended for implementation-dependent purposes and will not be defined by future versions of the CBEA. Unused fields and registers should be handled in the same manner as reserved fields and registers.

For descriptions of the implementation-dependent fields and registers, see the specific implementation documentation.

User Mode Environment

The *User Mode Environment (UME)* section defines the instruction set, base command set, storage models, and facilities available to an application programmer, as well as compatibility with the PowerPC Architecture. In addition to an overview, this section includes:

2. Overview

The *Introduction to Cell Broadband Engine Architecture* on page 19 provides a general review of the structure of a CBEA-compliant system. The reader should become familiar with that section and with *Figure 1-1 CBEA-Compliant Processor System* on page 20 to gain an understanding of the PowerPC Processor Element (PPE), the Synergistic Processing Unit (SPU), and the Memory Flow Controller (MFC) components, the system local storage and main storage memory arrays, their associated cache structures, and their organization within an overall system context.

The instructions and facilities provided by the PPE are defined in *PowerPC Architecture, Books I-III*.
The instructions and facilities provided by the SPU are defined in the *Synergistic Processor Unit Instruction Set Architecture* document. For a more complete understanding of the CBEA, the reader should also become familiar with these documents.

2.1 Instruction and Command Classes

Both the PPE and the SPU components execute programs that consist of instructions that specify the type of actions they are to perform. The MFCs execute commands that specify the type of data copying or movement they are to perform. Thus, there are generally two sets of instructions (PPE and SPU) and one set of commands (MFC). These instructions and commands can be categorized into three classes, as follows:

- Defined Class (see page 28)
- Illegal Class (see page 28)
- Reserved Class (see page 29)

The class of an instruction or command is determined by examining the opcode, and if it exists, the extended opcode. If an instruction opcode, or a combination of opcode and extended opcode, is not that of a defined or reserved instruction, then the instruction is illegal. If the command type is not that of a defined or reserved command, then the command is illegal.

A given instruction or command is in the same class for all implementations compliant with this release of the CBEA. In future versions of the CBEA, instructions or commands that are currently illegal can become defined (by being added to the architecture), or reserved (by being assigned to a special-purpose operation). Similarly, some instructions or commands that are currently reserved can become defined in a subsequent architecture release.

2.1.1 Defined Class

This class of instructions and commands contains all instructions and commands defined in this release of the CBEA. In general, defined instructions and commands are guaranteed to be provided in all implementations. The only deviations permitted are instructions or commands specifically identified in their descriptions as optional. Defined instructions or commands can have preferred forms, or invalid forms, or both. These forms are also indicated in the relevant description.

2.1.2 Illegal Class

This class of instructions and commands contains two sets of instructions and one set of commands: illegal PPE instructions, illegal SPU instructions, and illegal MFC commands. Illegal PPE instructions are described in *PowerPC Architecture, Book I*. Illegal SPU instructions are described in the *Synergistic Processor Instruction Set Architecture*. Illegal MFC commands are described in *Section 7.1.2 Illegal Commands* beginning on page 52. Illegal instructions and commands are available for use in future extensions of the CBEA. This means that a future release of the CBEA might assign any of these operation codes to new instructions or functions.

Any attempt to execute an illegal PPE instruction causes an exception interrupt, but has no other effect on the PPE operation. Any SPU that encounters an illegal instruction immediately halts program execution, records the event in its status register, and requests an external interrupt. The illegal-instruction interrupt should be enabled and routed to a PPE. In either case, the exception interrupt should cause the illegal-instruction handler for the system to be invoked, which then takes appropriate action.

Any PPE instruction that consists entirely of binary zeros is always guaranteed to be illegal. In an SPU instruction, an opcode of zero is a stop instruction, which causes SPU execution to stop. This increases the probability that any attempt to execute data or to uninitialize storage invokes the system illegal-instruction interrupt handler.

The MFC commands are not fully checked when enqueued. Therefore, an illegal MFC command might only be recognized as it is taken off the queue for execution, asynchronously to when it was enqueued. Consequently, illegal MFC commands are included with and treated in the same manner as, other causes of MFC or DMA asynchronous exceptions (see *Section 7.2* beginning on page 52).

In general, all exceptions, including illegal MFC commands and other DMA processing errors, cause the associated command queue processing to suspend as soon as they are detected. The exceptions also cause an exception interrupt to be generated and sent to a PPE. *Section 2.3 Exceptions* beginning on page 31 gives an overview of the various ways in which these exception interrupts can be generated.

2.1.3 Reserved Class

This class contains two sets of instructions and one set of commands. Reserved PPE instructions are defined in *PowerPC Architecture, Book I*. The reserved SPU instructions are defined in *Synergistic Processor Unit Instruction Set Architecture.* The reserved MFC commands are defined in *Section 7 MFC Commands* beginning on page 47. Reserved instructions are allocated to specific purposes outside the scope of the CBEA, or are intended for use in future extensions of the CBEA. These are the only commands that should be used by implementation-dependent applications. The specific implementation documentation describes how attempts to execute reserved instructions and commands are handled. If the use of a reserved instruction or command is not covered in the specific implementation documentation, it is treated as an illegal command.

2.2 Forms of Defined Instructions and Commands

In the defined set of instructions and commands, certain field or parameter settings can execute more efficiently, or can produce an error condition. The CBEA defines the field and parameter settings as preferred forms or invalid forms.

2.2.1 Preferred Forms

Some defined instructions and commands have preferred forms. The preferred form of an instruction or command executes in an efficient manner; any other form can take significantly longer to execute. Preferred PPE instruction forms are defined in *PowerPC Architecture, Book I*. The preferred forms of SPU instructions are defined in the *Synergistic Processor Unit Instruction Set Architecture* document. The preferred DMA MFC commands are defined in *Section 7 MFC Commands* beginning on page 47.

2.2.2 Invalid Forms

Some defined instructions and commands have invalid forms. An instruction or command is considered invalid if one or more fields (excluding those that specify the operation) are coded in a manner that can be deduced as incorrect by examining that encoding. Invalid PPE instruction forms are defined in *Section 4*

PowerPC Processor Element beginning on page 39. Invalid forms of SPU instructions are defined in *PowerPC Architecture, Book I*. Invalid DMA MFC commands are defined in *Section 7.1 Command Classes* beginning on page 49.

2.2.3 Optional Forms

Some of the defined instructions are optional. Any attempt to execute an optional instruction that is not provided by the implementation causes the system illegal-instruction interrupt handler to be invoked.

A facility, instruction, or command is optional for any of the following reasons:

1. It is being phased into the architecture. At some future date, it will be required and no longer optional.
2. It is being phased out of the architecture. System developers should develop a migration plan to eliminate its use in new systems.
3. It is useful primarily for certain kinds of applications and systems. It is likely to remain in the architecture, as optional.

Categories 1 and 2 permit the architecture to evolve gradually, by providing an intermediate status for facilities and instructions that are being added to or removed from the architecture. Category 3 is intended for facilities and instructions that are typically used primarily in library routines.

Currently, there are no optional MFC commands or instructions, but there is an optional facility, the Isolation Facility.

2.2.4 Optional Fields

Optional fields in the MFC commands are assumed to be zero if not explicitly set. Software does not have to set the optional fields if zeros achieve the desired results. For a detailed description of the MFC commands, see *Section 7 MFC Commands* beginning on page 47.

2.3 Exceptions

Note: For a more detailed description of exceptions, see the specific implementation documentation.

Exceptions are the result of an operation that cannot be executed as requested. In the CBEA, there are four types of exceptions:

- Exceptions caused directly by the execution of a PPE instruction
- Exceptions caused by the execution of an SPU instruction
- Exceptions caused by the execution of a MFC DMA command
- System-caused, asynchronous, external-event exceptions.

An exception can set status information in a register, and can cause an interrupt handler of the system software in the PPE to be invoked.

Exceptions caused by the execution of a PPE instruction are defined in *PowerPC Architecture, Book I*. In the *PowerPC Architecture*, there are only two types of exceptions: those caused by the execution of a PPE instruction and those caused by an asynchronous event. In most cases, the invocation of an interrupt handler for exceptions caused by the execution of a PPE instruction are precise (that is, the exception is created when the event happens, and the PPE instruction that caused the exception is known.) This is not true for floating-point exceptions, when the floating-point exceptions mode is set to one of the imprecise modes. The invocation of an interrupt handler for asynchronous events is always imprecise (see *PowerPC Architecture, Book I* for more information on PPE instruction-related exceptions).

Exceptions caused by the execution of an SPU instruction are defined in the *Synergistic Processor Unit Instruction Set Architecture* document and *Section 21 Interrupt Facilities* beginning on page 237. Exceptions caused by the execution of an MFC command are defined in *Section 21 Interrupt Facilities* beginning on page 237. These exceptions generate interrupts in the CBEA and are typically sent to the PPE as an external interrupt where they invoke a privileged software interrupt handler.

Exceptions caused by the execution of an SPU instruction, by the execution of an MFC command, or by asynchronous events are typically routed to the PPE and handled as imprecise external interrupts.

Exceptions generated directly by the execution of an instruction include:

- An attempt to execute an illegal instruction
- An attempt to execute a privileged instruction from the user mode environment (PPE only) (see *PowerPC Architecture, Book I* for more information on PPE instruction-related exceptions).
- The execution of a defined instruction using an invalid form
- The execution of an optional instruction not supported by the implementation
- An attempt to access storage with an effective address alignment that is invalid for the instruction (PPE only)
- The execution of a system-call instruction (PPE only)
- The execution of a trap instruction (PPE only)
- The execution of a floating-point instruction that causes a floating-point exception that is enabled (PPE only)
- The execution of a floating-point instruction that requires assistance from system software (PPE only)
- The execution of an interrupt mailbox channel write instruction by the SPU
- The execution of an SPU stop-and-signal instruction

The exceptions generated by an MFC command include:

- An attempt to execute an illegal MFC command
- An attempt to execute a defined MFC command using an invalid form (that is, invalid parameters)
- An attempt to execute a defined MFC command with an alignment error
- The execution of an optional MFC command not supported by the implementation
- An attempt to access storage not defined by the MFC-translation facility

2.4 SPU Events

The SPU supports an event facility that provides the capability to mask and to unmask events, wait on events, poll for events, and to provide interrupts for specific events. If the SPU interrupts are enabled an occurrence of an unmasked event results in an SPU interrupt handler being invoked with the first instruction of the interrupt handler located at local storage address ‘0’.

For more details, see *Section 9.11* beginning on page 133. Also refer to the *Synergistic Processor Unit Instruction Set Architecture* for information on enabling, disabling, and handling for program code executing on an SPU.

3. Storage Models

The CBEA-compliant processor implements two concurrent storage models for an application program: the virtual storage model of the PPE (also used by MFCs for DMA operations), and the local storage model of the SPU. The PPE virtual storage model allows privileged software to provide different views of the real memory and I/O devices for the PPE and any MFC unit DMA transfers. It is possible for multiple virtual address spaces to exist. The SPU local storage model is restricted to applications running on SPUs and data transfers handled by the MFC.

3.1 Virtual Storage Model

The virtual storage model implements a virtual storage model for applications. This means that a combination of hardware and software can present a storage model that allows applications to exist within a virtual address space larger than either the effective address space or the real address space.

Each program can access 2^{64} bytes of effective address space, subject to limitations imposed by privileged software. In a typical CBEA-compliant processor system, the effective address space of each program is a subset of a larger virtual address space managed by privileged software.

Each effective address is translated to a real address (an address of a byte in real storage or a byte on an I/O device) before being used to access storage. The hardware uses the address translation facility described in *PowerPC Architecture, Book III* to accomplish this. The privileged software manages the real storage resources of the system by setting up the tables and other information used by the hardware address translation facility.

The User Mode Environment deals primarily with effective addresses that are translated by the address translation facility. Each effective address lies in a virtual page[1], which is mapped to a real page (4-KB virtual page) or to a contiguous sequence of real pages (large virtual page) before data or instructions in the virtual page are accessed.

In general, real storage might not be large enough to map all the virtual pages used by active applications. With hardware support, the privileged software can attempt to use the available real pages to map a set of virtual pages that is sufficient for the applications. If a sufficient set of virtual pages is maintained, “paging” activity is minimized. If sufficient virtual pages are not available, performance degradation is likely.

Based on system standards and application requests, the privileged software can restrict access to virtual pages. Access to the virtual pages can be read/write, read only, or no access. For example, program code might be designated read only.

Refer to *PowerPC Architecture, Book III* for a complete description of the virtual storage model.

[1] Page: An aligned unit of storage for which protection and control attributes can be specified independently, and for which reference and change status are independently recorded.

3.2 SPU Local Storage Model

Each SPU has its own dedicated area of local storage. Applications running on a given SPU can only reference the associated local storage area using a local address for instruction fetch, data load, and data store operations. The individual local storage areas can be aliased to a real address within the main storage domain and any PPE can access these areas by using the appropriate effective address.

MFC units process data transfers, which move data between a local address and an effective address. The local address always references the local storage area associated with the MFC, while the effective address can be arranged to reference any area in the main storage domain, including aliased local storage areas, if required.

3.2.1 Local Storage Access

The CBEA allows the local storage of an SPU to have an alias in the real address space in the main storage domain. This allows other processors in the main storage domain to access local storage through appropriately mapped effective address space. It also allows external devices, such as a graphics device, to directly access the local storage.

3.2.1.1 Mapping Requirements

Privileged software should access the aliased pages of local storage in the main storage domain. If not accessed as caching inhibited, software must explicitly manage the coherency of local storage with other system caches.

3.2.1.2 Local Storage Access Exceptions

MFC commands, which access an effective address range that maps to its own local storage can produce an error or unpredictable results. This occurs when the translated effective address area for a DMA overlaps the local storage range of a DMA. If the two address ranges (translated effective address and local storage address) overlap and the source is a lower address than the destination, the DMA results in the corruption of the source data. Address overlap is not detectable and does not generate an exception. Therefore, it is the programmer's and privileged software's responsibility to avoid an unintended overlap, which can result in the corruption of data.

3.3 Single-Copy Atomicity

An access is single-copy atomic, or simply atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are thus serialized; each happens in its entirety in some order, even when that order is not specified in the program or enforced between processors. In the PowerPC Architecture, the following single register accesses are always atomic:

- Byte accesses (all bytes are aligned on byte boundaries)
- Halfword accesses aligned on halfword boundaries
- Word accesses aligned on word boundaries
- Doubleword accesses aligned on doubleword boundaries
- Quadword accesses aligned on quadword boundaries

No other accesses are guaranteed to be atomic.

An access that is not atomic is performed as a set of smaller, disjointed atomic accesses. The number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed.

In the CBEA, DMA accesses in the main storage domain are atomic if they meet the requirements of the PowerPC Architecture. All other DMA transfers, if greater than a quadword or unaligned, are performed as a set of smaller, disjointed atomic accesses. The number and alignment of these accesses are implementation-dependent, as is the relative order in which they are performed. Only quadword accesses of local storage are atomic.

3.4 Cache Models

A cache model in which there is one cache for instructions and another cache for data is called a "Harvard-style" cache. This is the model assumed by the PowerPC Architecture. Alternative cache models can be implemented (such as, a "combined cache" model, in which a single cache is used for both instructions and data, or a model in which there are several levels of caches), but they must support the programming model implied by a Harvard-style cache.

The processor is not required to maintain copies of storage locations in the instruction cache consistent with modifications to those storage locations (such as, modifications caused by store instructions). A location in the data cache is considered to be modified in that cache if the location has been modified (for example, by a store instruction) and the modified data has not been written to main storage.

Cache management instructions allow programs to manage the caches when needed. For example, program management of the caches is needed when a program generates or modifies code that is executed (that is, when the program modifies data in storage and then attempts to execute the modified data as instructions). The cache management instructions are also useful in optimizing the use of memory bandwidth in such applications as graphics and numerically intensive computing. The functions performed by these instructions depend on the storage control attributes associated with the specified storage location.

The Cache Management Instructions allow programs to:

- Invalidate the copy of storage in an instruction cache block (**icbi**)
- Provide a hint that the program will probably soon access a specified data cache block (**dcbt**, **dcbtst**)
- Set the contents of a data cache block to zeros (**dcbz**)
- Copy the contents of a modified data cache block to main storage (**dcbst**)
- Copy the contents of a modified data cache block to main storage and make the copy of the block in the data cache invalid (**dcbf**)

The SL1 data cache commands allow programs to:

- Bring a range of effective addresses into the SL1 (**sdcrt**)
- Bring a range of effective addresses into the SL1 (**sdcrtst**)
- Write zeros to the contents of a range of effective addresses (**sdcrz**)
- Store the modified contents of a range of effective addresses (**sdcrst**)
- Store the modified contents of a range of effective addresses and invalidate the block (**sdcrf**)

Note: These instructions are treated as no operations (no-op) instructions in implementations without an SL1.

3.5 Memory Coherence

Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are coherent if they are serialized in some order, and no processor or no device is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the real storage location need not assume each of the values written to it. For example, a processor can update a location several times before the value is written to real storage.

The result of a store operation is not available to every processor or to every device at the same instant; a processor or device can observe only some of the values that are written to a location. However, when a location is accessed atomically and coherently by all processors and devices, the sequence of values loaded from the location by any processor or any device during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or device can never load a “newer” value first and then, later, load an “older” value.

Memory coherence is managed in blocks called coherence blocks. Their size is implementation-dependent, but is usually larger than a word and often the size of a cache block.

For storage that does not require memory coherence, software must explicitly manage memory coherence to the extent required by program correctness. The operations required to do this can be system dependent.

Programming Note:

In most systems the default is that all storage is memory coherence required. For some applications in some systems, software management of coherence can yield better performance. In such cases, a program can request that a given unit of storage not be memory coherence required, and can manage the coherence of that storage by using the **sync** instruction, the cache management instructions, and services provided by the operating system.

3.6 Storage Control Attributes

Storage control attributes are associated with units of storage that are multiples of the page size. Each storage access is performed according to the storage control attributes of the specified storage location. The storage control attributes are:

- Write through Required
- Caching Inhibited
- Memory Coherence Required
- Guarded

These attributes have meaning only when an effective address is translated by the processor performing the storage access. All combinations of these attributes are supported except Write through Required with Caching Inhibited.

Programming Note:

The Write through Required and Caching Inhibited attributes are mutually exclusive because the Write through Required attribute permits the storage location to be in the data cache while the Caching Inhibited attribute does not. Storage that is Write through Required or Caching Inhibited is not intended to be used for general-purpose programming. For example, the **lwarx**, **ldarx**, **stwcx.**, and **stdcx.** instructions can cause the interrupt handler for system data storage to be invoked if they specify a location in storage having either of these attributes.

3.7 Shared Storage

The CBEA supports the sharing of storage between programs, between different instances of the same program, between SPUs, and between processors and other devices. It also supports access to a storage location by one or more programs using different effective addresses or DMA addresses. All these cases are considered storage sharing. Storage is shared in blocks of an integral number of pages.

When the same storage location has different effective addresses, the addresses are called aliases. Each application can be granted separate access privileges to aliased pages.

4. PowerPC Processor Element

The CBEA includes a PowerPC processor, which, with the MFC is known as the PowerPC Processor Element (PPE). The architecture of the PowerPC Processor Element is defined in *PowerPC Architecture, Books I- III*.

The PPE must be a 64-bit implementation, in which all effective addresses and registers, except some special-purpose and memory-mapped I/O (MMIO) registers are 64 bits long. All implementations have two modes of operation: 64-bit mode and 32-bit mode. The mode controls how the effective address is interpreted, how status bits are set, and how the count register is tested by branch-conditional instructions. All instructions are available in both modes. In both 64-bit mode and 32-bit mode, instructions that set a 64-bit register affect all 64 bits, and the value placed in the register is independent of mode. In both modes, effective address computations use all 64 bits of the relevant registers (such as, the general-purpose registers, link register, and count register) and produce a 64-bit result. However, in 32-bit mode, the high-order 32 bits of the computed effective address are ignored when accessing data and are set to zero when fetching instructions.

The CBEA does not permit a PPE implementation that provides only the equivalent of 32-bit mode (an implementation in which all registers except floating-point registers are 32 bits long).

4.1 PowerPC Architecture Book I and Book II Compatibility

The CBEA User Mode Environment is compatible with of *PowerPC Architecture, Book I* and the CBEA Privileged Mode Environment is compatible with *PowerPC Architecture, Book II.* The PPE provides binary compatibility for PowerPC applications, except as described in *Section 4.1.2 Incompatibilities with PowerPC Architecture, Book I* on page 39.

The CBEA does *not* automatically track changes and extensions to the PowerPC Architecture. Inclusion of changes and extensions to the PowerPC Architecture will be identified in future version of the CBEA.

4.1.1 Optional Features in PowerPC Architecture, Book I (Required for CBEA)

The following facilities and instructions are considered optional in the PowerPC Architecture, but are required for the PPE by the CBEA user mode environment.

- Floating reciprocal estimate single A-form (**fres**)
- Floating reciprocal square-root estimate A-form (**frsqte**)
- Vector/SIMD multimedia extension

Note: The optional PowerPC floating-point instructions that are mandatory in the CBEA are needed for the application space targeted by the CBEA.

4.1.2 Incompatibilities with PowerPC Architecture, Book I

Currently there are no incompatibilities with *PowerPC Architecture, Book I*.

4.1.3 Optional Features in PowerPC Architecture, Book II (Required for CBEA)

The following facilities and instructions are considered optional in the PowerPC Architecture, but are required in the CBEA.

- Data cache block touch X-form (**dcbt**)

  This is an optional version of **dcbt** that permits a program to provide a hint that a sequence of data cache blocks is likely to be needed soon.

Book II describes these facilities and instructions.

4.1.4 Incompatibilities with PowerPC Architecture, Book II

Currently there are no incompatibilities with *PowerPC Architecture, Book II*.

4.1.5 Extensions to the PowerPC Architecture

For information on extensions in the CBEA to the PowerPC Architecture, see *Appendix E* .

5. Synergistic Processor Unit

The intent of the SPU is to fill a void between general-purpose processors and special-purpose hardware. Where general-purpose processors aim to achieve the best average performance on a broad set of applications, and special-purpose hardware aims to achieve the best performance on a single application, the SPU aims to achieve leadership performance on critical workloads for game, media, and broadband systems. The intent of the SPU and the CBEA is to provide a high degree of control to expert (real-time) programmers while maintaining ease of programming.

The SPU implements a new instruction set architecture (ISA). This architecture is described in a separate document, the *Synergistic Processor Unit Instruction Set Architecture*.

The main characteristics of this architecture are:

- Load-and-store architecture with sequential semantics, using a set of 128 registers, each of which is 128 bits wide.
- Single-instruction, multiple-data (SIMD) capability
  - Sixteen 8-bit integers
  - Eight 16-bit integers
  - Four 32-bit integer or four single-precision floating-point values
  - Two double-precision floating point
- Load-and-store access to an associated local storage.
- Channel input/output for MFC control (used for external data access).

The SPU has the following restrictions:

- No direct access to main storage (access to main storage using MFC facilities only)
- No distinction between user mode and privileged state
- No access to critical system control such as page-table entries (this restriction should be enforced by PPE privileged software).
- No synchronization facilities for shared local storage access

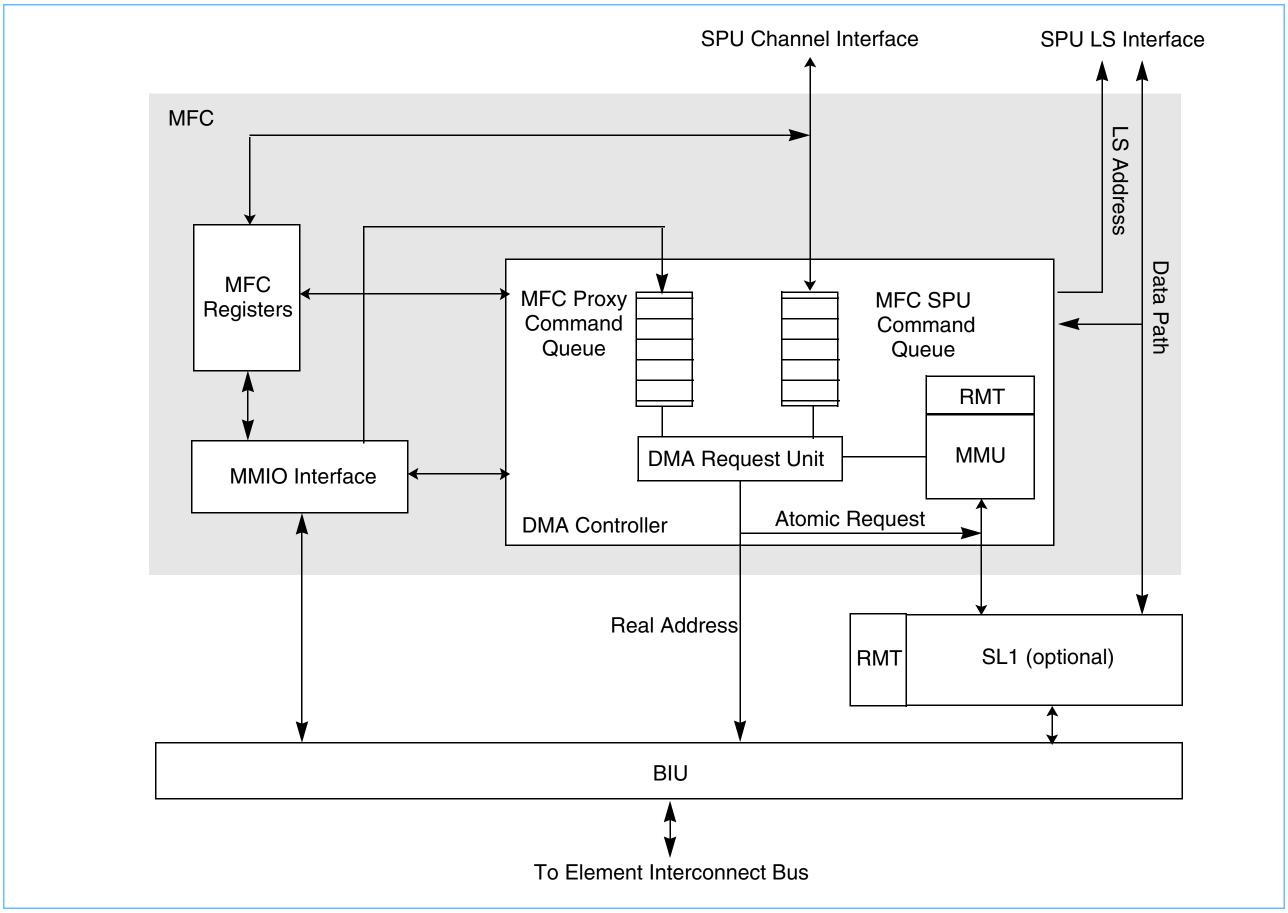
The intent of the SPU is to enable applications that require a high computational unit density, and can effectively make use of the provided instruction set. A significant number of SPU cores in a system, managed by the PPE, allow for cost-effective processing over a wide range of applications.

6. Memory Flow Controller

In a CBEA-compliant processor, the MFC serves as an interface to the system and to other elements for an SPU. It provides the primary mechanism for data transfer, protection, and synchronization between main storage and the local storage arrays. As mentioned earlier, there is logically an MFC for each SPU in a processor. Some implementations can share resources of a single MFC between multiple SPUs. In this case, all the facilities and commands defined for the MFC must appear independent to software for each SPU. The effects of sharing an MFC must be limited to implementation-dependent facilities and commands.

Figure 6-1 shows a high-level block diagram of a typical MFC. In this illustration, the MFC has two interfaces to the SPU, two interfaces to the Bus Interface Unit (BIU), and two interfaces to an optional SL1 cache. The SPU interfaces are the SPU channel interface and the SPU local storage interface. The SPU channel interface allows the SPU to access MFC facilities and to issue MFC commands. The SPU local storage interfaces is used by the MFC to access the local storage in the SPU. One interface to the BIU allows memory-mapped I/O (MMIO) access to the MFC facilities. This interface also allows other processors to issue MFC commands. Commands issued using MMIO are referred to as MFC proxy commands. The other interface to the BIU carries the real address. The interfaces to the SL1 cache are mainly for data transfers. One interface is used by the MFC for access to the address translation tables in main storage and the other interface of the SL1 cache is used for the transfer of data between main storage and local storage.

Figure 6-1. Typical MFC Block Diagram



As shown in *Figure 6-1*, the following are the main units in a typical MFC:

- MMIO interface
- MFC registers
- DMA controller

The MMIO interface maps the MFC facilities of the SPU into the real address space of the system. This allows access to the MFC facilities from any processor, or any device in the system. In addition, the MMIO interface can be configured to map the local storage of the SPU into the real address space. This allows direct access to the local storage from any processor or any device in the system, enabling local-store-to-local-store transfers and the ability for I/O devices to directly access the local storage domain of an SPU. Coherency is not maintained between SPU and MMIO accesses of the local storage domain.

6.1 MFC Facilities

Most of the MFC facilities are contained in the MFC Registers unit. Some facilities are contained in the Direct Memory Access Controller (DMAC). Below is a list of the facilities within the MFC. These facilities are described in other sections of this document.

User mode environment facilities include:

- Mailbox Facility (see page 90)
- SPU Signal Notification Facility (see page 94)
- Proxy Tag-Group Completion Facility (see page 82)
- MFC Multisource Synchronization Facility (see page 96)
- SPU Control and Status Facilities (see page 86)
- SPU Isolation Facility (see page 163)

Privileged mode environment facilities include:

- MFC Privileged Facilities (see page 197)
  - MFC State Register One (see page 197)
  - MFC Logical Partition ID Register (see page 199)
  - MFC Storage Description Register (see page 200)
  - MFC Data Address Register (see page 201)
  - MFC Data Storage Interrupt Status Register (see page 202)
  - MFC Address Compare Control Register (see page 203)
  - MFC Local Storage Address Compare Facility (see page 205)
  - MFC Command Error Register (see page 207)
  - MFC Data Storage Interrupt Pointer Register (see page 208)
  - MFC Control Register (see page 209)
  - MFC Atomic Flush Register (see page 212)
  - SPU Outbound Interrupt Mailbox Register (see page 213)
- SPU Privileged Facilities (see page 215)
  - SPU Privileged Control Register (see page 215)
  - SPU Local Storage Limit Register (see page 217)
  - SPU Configuration Register (see page 221)
- SPE Context Save and Restore (see page 223)

The synchronization and the transfer of data is generally the responsibility of the DMAC within the MFC. The DMAC can move data between the local storage of an SPU and the main storage area. Optionally, the data can be cached in the SL1.

The SPEs and PPE instruct the MFC to perform these DMA operations by queuing DMA command requests to the MFC through one of the command queues:

- Commands issued by an SPE are queued to the MFC SPU command queue
- Commands issued by a PPE are queued to the MFC proxy command queue

The MFC uses a MMU to perform all MFC address translations and MFC access protection checks required for the DMA transfers. The MMU handles MFC transfers in much the same way that the PPE storage addressing facility handles load-and-store operations.

Ordering of the data transfers for a given command adheres to the rules specified in the PowerPC Architecture. The ordering between commands is described in *Section 7.9 MFC Synchronization Commands* beginning on page 62*.*

The memory management facilities of a CBEA-compliant processor are described in *Section 14 Storage Addressing* beginning on page 177.

Section 7.5 Get Commands (Main Storage to Local Storage) beginning on page 54 and *Section 7.6 Put Commands (Local Storage to Main Storage)* beginning on page 56 describe the DMA data transfer commands available to the application programmer. The MFC employs a 64-bit effective address field, which is translated and then used to reference main storage. A 32-bit address field is used to reference local storage directly.

7. MFC Commands

MFC commands provide the main method that enables code executing in an SPU to access main storage and maintain synchronization with other processors and devices in the system. Commands are also provided to manage optional caches.

MFC commands can either be issued by code running on the SPU, or by code running on another processor or device, such as the PPE. Code running on the associated SPU executes a series of channel instructions to issue an MFC command. Code running on other processors or devices performs a series of memory-mapped I/O (MMIO) transfers to issue an MFC command to an SPE.

The commands issued are queued to one of these command queues of the MFC:

- MFC proxy command queue for any MMIO-initiated commands
- MFC SPU command queue for any channel-initiated commands

In general, commands can be queued using the MMIO registers, or through channel instructions executed by the associated SPU. The MMIO method is intended for use by the PPE to control the transfer of data between main storage and the associated local storage on behalf of the SPE.

MFC commands that transfer data are referred to as MFC DMA commands. The data transfer direction for MFC DMA commands is always referenced from the perspective of an SPE. Therefore, commands that transfer data into an SPE (from main storage to local storage) are considered **get** commands, while commands that transfer data out of an SPE (from local storage to main storage) are considered **put** commands.

The following suffixes are associated with the MFC DMA commands, and extend or refine the function of a command. For example, a **put** command moves data from local storage to the effective address within the main storage domain. A **puts** command moves data from local storage to the effective address within the main storage domain and starts the SPU after the DMA operation completes.

s Starts the execution of the SPU at the current location indicated by the SPU Next Program Counter Register after the data has been transferred into or out of the local storage.

r Performance hint for DMA put operations. The hint is intended to allow another processor or device, such as the PPE, to capture the data into its cache.

f Tag specific fence. Commands with a tag-specific fence are locally ordered with respect to all previously-issued commands within the same tag group and command queue.

b Tag specific barrier. Commands with a tag-specific barrier are locally ordered with respect to all previously-issued commands within the same tag group and command queue and all subsequently-issued commands to the same command queue with the same tag.

l List command. Executes a list of DMA list elements located in local storage. The maximum number of elements is 2048, and each element describes a transfer of up to 16K bytes.

Commands with an “s” suffix can only be issued to the MFC proxy command queue. Commands with an “l” suffix and all the MFC atomic commands can only be issued to the MFC SPU command queue. All other commands described in this section can be issued to either of the command queues. Commands issued from a PPE are issued for an SPE and are sent to the MFC proxy command queue.

The descriptions of the MFC commands are shown in *Section 7.3 MFC Command Parameters* beginning on page 53. Each description shows the command mnemonic followed by a list of the mnemonics for the parameters that affect the operation of that command.

cmd CL, TG, TS/LSZ, LSA, [EAH,] EAL/LA

The parameter mnemonics are listed in *Table 7-1*. As shown, not all parameters are used by all commands. Optional parameters, such as EAH, are enclosed in brackets. (When EAH is not specified on a command, EAH must be set by hardware to ‘0’.)

Table 7-1. Parameter Mnemonics

Parameter	Parameter Name	Register Name	See Note
CL	MFC Class ID	MFC_ClassID	
TG	MFC Command Tag Identification	MFC_Tag	
TS	MFC Transfer Size	MFC_Size	1
LSZ	MFC List Size	MFC_Size	1
LSA	MFC Local Storage Address	MFC_LSA	
EAH	MFC Effective Address High	MFC_EAH	4
EAL	MFC Effective Address Low	MFC_EAL	2
LA	MFC List Local Storage Address	MFC_EAL	2
LTS	List Element Transfer Size		3
LEAL	List Element Effective Address Low		3

1. TS and LSZ share the same register offset. The meaning of the contents depends on the suffix of the MFC opcode.
2. EAL and LA share the same register offset. The meaning of the contents depends on the suffix of the MFC opcode.
3. No associated registers. These parameters are located in local storage and are referenced by the list address (LA) parameter
4. This parameter is optional

Notes:

(For more information on the following code, see *Conventions and Notation* on page 17.)

1. A list command is equivalent to performing a series of commands where the opcode is that of the command without the “l” suffix. Each element in an MFC list defines one of the series of MFC transfers to perform. For example:

```
getl CL, TG, LSZ, LSA, [EAH], LA is equivalent to:
{
   get CL, TG, LA (LSZ), LSA, [EAH], LA (LEAL)
   /* LSA is moved to next 16-byte boundary in LSW
   LSA = LSA + LA (LSZ);
   if (LSA% 16!= 0)
     LSA = (LSA & x’7FF0’ + 16)
     LA = LA + 8
}
```

2. The effective address (EA) of an MFC command is formed by:
    - For non-list type commands: **(EAH << 32) | EAL**
    - For list type commands: **EAH << 32) | LA (LEAL**), where **LA (LEAL)** in the effective address of the current list element pointed to by the list address. This construct assumes the **LA** is incremented after each transfer.
3. The size of a transfer is **TS** bytes for non-list commands and **LA(LTS)** bytes for each element of a list command. For a list command, the number of elements is **LSZ** divided by 8 bytes (that is, 8 bytes per list element).
4. See *Section 7.9 MFC Synchronization Commands* on page 62, for a description of the barrier suffixes.

7.1 Command Classes

Commands can be categorized into three classes, as follows:

- Defined
- Illegal
- Reserved

The class of a command is determined by examining the opcode, and if it exists, the extended opcode. If a command opcode, or a combination of opcode and extended opcode is not that of a defined or reserved command, then the command is illegal.

A given command is in the same class for all implementations compliant with this release of the CBEA. In future versions of the CBEA, commands that are currently illegal might become defined (by being added to the architecture), or reserved (by being assigned to a special-purpose operation). Similarly, some commands that are currently reserved might become defined in a subsequent architecture release.

7.1.1 Defined Commands

Defined commands fall into one of three categories:

- Data transfer commands (see *Table 7-2 Data Transfer or MFC DMA Commands* on page 50)
    - Data moved from local storage and placed in main storage (**put** commands)
    - Data moved into local storage from main storage (**get** commands)
- SL1 cache-management commands (see *Table 7-3 SL1 Storage Control Commands* on page 51)
- Synchronization commands (see *Table 7-4 MFC Synchronization Commands* on page 51)

The data transfer commands are further divided into sub-categories that define the direction of the data movement (that is, to or from local storage). An application can place the data transfer commands listed in *Table 7-2* on page 50, into the MFC proxy command queue. Unless otherwise noted, these commands can be executed in any order (asynchronous).

Note: Embedded barrier, fence, and synchronization commands *must* be used to ensure proper ordering when ordering is required.

Table 7-2. Data Transfer or MFC DMA Commands (Page 1 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
Put Commands			
put	x‘0020’	Proxy/Channel	Moves data from local storage to the effective address within the main storage domain.
puts	x‘0028	Proxy	Moves data from local storage to the effective address within the main storage domain and starts the SPU after the DMA operation completes.
putr	x‘0030’	Proxy/Channel	Same as **put** with a PPE L2 cache scarf hint (used to send results to the PPE). (Scarfing is the direct transfer of data to the PPE L2 cache.)
putf	x‘0022’	Proxy/Channel	Moves data from local storage to the effective address within the main storage domain with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putb	x‘0021’	Proxy/Channel	Moves data from local storage to the effective address within the main storage domain with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putfs	x‘002A’	Proxy	Moves data from local storage to the effective address within the main storage domain with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after the DMA operation completes.
putbs	x‘0029’	Proxy	Moves data from local storage to the effective address within the main storage domain with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after DMA operation completes.
putrf	x‘0032’	Proxy/Channel	Same as **putf** with a PPE L2 cache scarf hint (used to send results to the PPE). (Scarfing is the direct transfer of data to the PPE L2 cache.)
putrb	x‘0031’	Proxy/Channel	Same as **putb** with a PPE L2 cache scarf hint (used to send results to the PPE). (Scarfing is the direct transfer of data to the PPE L2 cache.)
putl	x‘0024’	Channel	Moves data from local storage to the effective address within the main storage domain using an MFC list.
putrl	x‘0034’	Channel	Same as **putl** with a PPE L2 cache scarf hint (used to send results to the PPE).
putlf	x‘0026’	Channel	Moves data from local storage to the effective address within the main storage domain using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	x‘0025’	Channel	Moves data from local storage to the effective address within the main storage domain. using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putrlf	x‘0036’	Channel	Same as **putlf** with a PPE L2 cache scarf hint (used to send results to the PPE).
putrlb	x‘0035’	Channel	Same as **putlb** with a PPE L2 cache scarf hint (used to send results to the PPE).
Get Commands			
get	x‘0040’	Proxy/Channel	Moves data from the effective address within the main storage domain to local storage.
gets	x‘0048’	Proxy	Moves data from the effective address within the main storage domain to local storage, and starts the SPU after DMA operation completes.
getf	x‘0042’	Proxy/Channel	Moves data from the effective address within the main storage domain to local storage with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).

Table 7-2. Data Transfer or MFC DMA Commands (Page 2 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
getb	x'0041'	Proxy/Channel	Moves data from the effective address to local storage with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
getfs	x'004A'	Proxy	Moves data from the effective address within the main storage domain to local storage with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after DMA operation completes.
getbs	x'0049'	Proxy	Moves data from the effective address within the main storage domain to local storage with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after DMA operation completes.
getl	x'0044'	Channel	Moves data from the effective address within the main storage domain to local storage using an MFC list.
getlf	x'0046'	Channel	Moves data from the effective address within the main storage domain to local storage using an MFC list with **fence** (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	x'0045'	Channel	Moves data from the effective address within the main storage domain to local storage using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Table 7-3 lists the available SL1 storage control commands.

Table 7-3. SL1 Storage Control Commands

Mnemonic	Opcode	Support	Description
sdcrt	x'0080'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA gets).[1]
sdcrtst	x'0081'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA puts).[1]
sdcrz	x'0089'	Proxy/Channel	Writes '0's to the contents of a range of effective addresses.
sdcrst	x'008D'	Proxy/Channel	Stores the modified contents of a range of effective addresses.
sdcrf	x'008F'	Proxy/Channel	Stores the modified contents of a range of effective addresses and invalidates the block.

[1] These commands do not transfer data in implementations without an SL1.

Table 7-4 lists the synchronization commands available in the CBEA.

Table 7-4. MFC Synchronization Commands

Command	Opcode	Support	Description
sndsig	x'00A0	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU. (This command is actually a 4-byte DMA put that can go to any address.)
sndsigf	x'00A2	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU with fence.(This command is actually a 4-byte DMA put that can go to any address.)
sndsigb	x'00A1	Proxy/Channel	Updates signal notification registers in an I/O device or another SPU with barrier. (This command is actually a 4-byte DMA **put** that can go to any address.)

Table 7-4. MFC Synchronization Commands

Command	Opcode	Support	Description
barrier	x’00C0	Proxy/Channel	Barrier type ordering. Ensures ordering of all preceding, nonimmediate DMA commands with respect to all commands following the **barrier** command within the same command queue. The **barrier** command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	x’00C8	Proxy/Channel	Controls the ordering of **get** commands with respect to **put** commands, and of **get** commands with respect to **get** commands accessing storage that is caching inhibited and guarded. Also controls the ordering of **put** commands with respect to **put** commands accessing storage that is memory coherence required and not caching inhibited
mfcsync	x’00CC	Proxy/Channel	Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and devices in the system

Table 7-5 lists the atomic commands available in the CBEA.

Table 7-5. MFC Atomic Commands

Command	Opcode	Support	Description
getllar	x’00D0	Channel	Gets lock line and creates a reservation (executes immediately).
putllc	x’00B4	Channel	Puts lock line conditional on a reservation (executes immediately).
putlluc	x’00B0	Channel	Puts lock line unconditional (executes immediately).
putqlluc	x’00B8	Channel	Puts lock line unconditional (queued form).

7.1.2 Illegal Commands

Illegal class of commands are intended for future versions of the CBEA. A future version of the CBEA might define an operation for command opcodes in the illegal class. The illegal class of commands is commands not in the defined class or in the reserved class. For this version of the CBEA, the illegal class of commands includes any command whose reserved upper 8 bits in the MFC command opcode parameter are in the range of x’0100 ≤ MFC command opcode ≤ x’7FFF.

7.1.3 Reserved Commands

Reserved commands are intended for implementation dependent use. Commands in this class have the reserved upper 8 bits of the MFC command opcode parameter in the range of x’8000 ≤ MFC command opcode ≤ x’8FFF.

7.2 Command Exceptions

Unaligned DMAs are not supported by the CBEA. If an unaligned DMA operation is encountered, the MFC command queue processing is suspended and an DMA alignment interrupt is generated. For a detailed explanation of alignment exceptions, see *Section 21 Interrupt Facilities* beginning on page 237.

7.3 MFC Command Parameters

The parameters are not the same for the PPE and SPU. The PPE does not support LSZ, LTS, or LA parameters. In addition, the L and S command modifiers are not supported on the PPE. The issue sequence and policy differences between the PPE and the SPU are described in *Section 8.2 MFC Proxy Command Issue Sequence* beginning on page 78 and in *Section 9.2 MFC SPU Command Issue Sequence* beginning on page 110. The command opcode is described in *Section 8.1.1 MFC Command Opcode Register* on page 71. The parameters are supported as described in the following sections:

Class parameter (***CL***)	See *Section 8.1.2 MFC Class ID Register* beginning on page 72.
Tag parameter (***TG***)	See *Section 8.1.3 MFC Command Tag Register* beginning on page 73.
Transfer size or list size parameters (***TS*** or ***LSZ***)	See *Section 8.1.4 MFC Transfer Size Register* beginning on page 74.
Local storage address parameter (***LSA***)	See *Section 8.1.5 MFC Local Storage Address Register* beginning on page 75.
Optional effective address high parameter	See *Section 8.1.6 MFC Effective Address High Register* beginning on page 76.
Effective address low address	See *Section 8.1.7 MFC Effective Address Low Register* beginning on page 77.

7.4 DMA List Elements

Commands with a suffix of “l” use list elements located in the local storage pointed to by the DMA list local storage address (LA) parameter of a list command. The element contains the lower order word of the effective address (LEAL) and the transfer size (LTS). The element also contains a stall-and-notify flag.

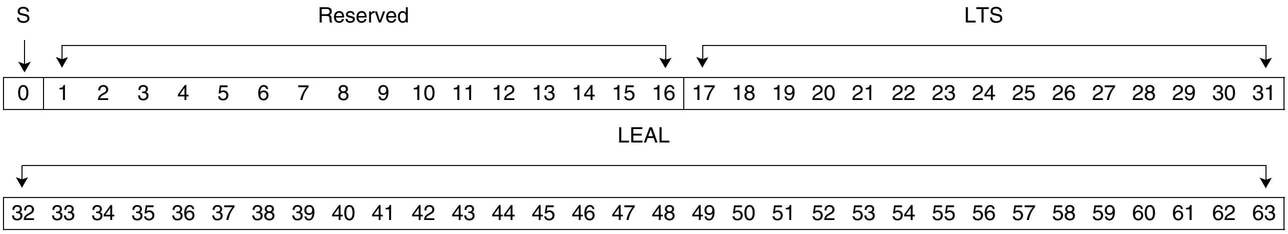
The DMA list commands use a list of effective addresses and transfer size pairs, or list elements, stored in local storage as the parameters for the DMA transfer. These parameters are used for SPU-initiated DMA list commands, which are not supported on the MFC proxy command queue. The first word contains the transfer size and a stall-and-notify flag. The second word contains the lower order 32 bits of the effective address. While the starting effective address is specified for each transfer element in the list, the local storage address involved in the transfer is only specified in the primary list command. The local storage address is internally incremented based on the amount of data transferred by each element in the list. However, due to alignment restrictions, if the local storage address does not begin on a 16-byte boundary for a list element transfer, the hardware automatically increments the local storage address to the next 16-byte boundary. This only occurs if transfer sizes less than 16 bytes are used. List elements with transfer sizes less than 16 bytes use a local storage offset within the current quadword (16 bytes) defined by the four least significant bits of the effective address.

Effective addresses specified in the list elements are relative to the 4-GB area defined by the upper 32 bits of the effective address specified in the base DMA list command. While DMA list starting addresses are relative to the single 4-GB area, transfers within a list element can cross the 4-GB boundary.

Setting the “S” (stall-and-notify) bit causes the DMA operation to suspend execution of this list after the current list element has been processed, and to set a stall-and-notify event status for the SPU. Execution of the stalled list does not resume until the MFC receives a stall-and-notify acknowledgment from the SPU program. Stall-and-notify events are posted to the SPU program using the associated command tag group identifier. When there are multiple DMA list commands in the same tag group with stall-and-notify elements, software should ensure that a tag-specific **barrier** or global **barrier** is used to force ordered execution of the DMA list commands to avoid ambiguity. Setting the “S” bit on the last element in a list is not supported. The “S” bit on the last element will be ignored.

All DMA list elements within the DMA list command are guaranteed to be started and issued in sequence. All elements within a DMA list command have an inherent local ordering. All transfers to guarded pages within a DMA list must adhere to the caching inhibited and guarded page semantics with respect to ordering.

A single DMA list command can contain up to 2048 elements, occupying 16 KB of local storage.



Bits	Field Name	Description
0	S	Stall-and-notify bit
1:16	Reserved	Reserved
17:31	LTS	List Element Transfer size (LTS)
32:63	LEAL	Low word of the 64-bit effective address (LEAL)

Programming Note:

If at all possible, list elements with transfer sizes of 128 bytes or more should be aligned on 128-byte boundaries for maximum performance. Smaller transfers and transfers not aligned on 128-byte boundaries result in poorer performance. Transfers of less than 16 bytes are inefficient and should be used only when necessary to interface with I/O devices.

7.5 Get Commands (Main Storage to Local Storage)

Get commands are conventional DMA transfer commands. The specified effective address is translated into a real address by the MMU as appropriate. Then transfer size bytes of data are copied starting from the translated real address to the destination local storage address.

When the **get** and **getf** commands have an “s” suffix, they can set the run bit after the local storage has been updated. Commands with an “s” suffix can only be executed from the MFC proxy command queue.

7.5.1 DMA Get Command

The get (**get**) command transfers the number of bytes specified by the transfer size parameter from the effective address to the local storage address of the corresponding SPU.

get ***CL, TG, TS, LSA, [EAH,] EAL***
gets ***CL, TG, TS, LSA, [EAH,] EAL***

7.5.2 DMA Get with Fence or with Barrier Command

Like the **get** command, the get with fence or with barrier (**get*<f,b>[s]***) command transfers the number of bytes specified by the transfer size parameter from the effective address to the local storage address of the corresponding SPU. Unlike the **get** command, the **get*<f,b>*** command provides local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID).

getf ***CL, TG, TS, LSA, [EAH,] EAL***
getb ***CL, TG, TS, LSA, [EAH,] EAL***
getfs ***CL, TG, TS, LSA, [EAH,] EAL***
getbs ***CL, TG, TS, LSA, [EAH,] EAL***

7.5.3 DMA Get List Command

The get list (**getl**) command provides software with a method to move discontinuous blocks of data in its effective address space to a contiguous area of local storage using a single DMA command. This command uses a list of effective address (low) or transfer size pairs (DMA list) stored in local storage as the source of the DMA operation. The list address parameter of this command contains the starting address of the DMA list in local storage. The list address is a local storage address, and is therefore not translated by the MMU. The number of bytes in the list is provided in the list size parameter. The list size parameter must be a multiple of 8 bytes for this DMA command, and the list address parameter must be aligned on an 8-byte boundary in local storage. The maximum list size is architecturally 16 KB. However, the supported list size is implementation-dependent. This command is not available from the MFC proxy command queue.

The DMA parameters provided in local storage must follow the same format as the **get** command. The local storage address (LSA) must start on a 16-byte boundary if the transfer size of the first list element is 16 bytes or less.

The effective address (high) parameter is provided as part of the **getl** command. While the starting effective address is always EAH || LEAL, a DMA transfer within a list element can cross the 232 boundary.

Note: A DMA command with list operation that uses a local storage destination within the list area produces unpredictable results, if the transfer modifies list elements not yet started.

getl ***CL, TG, LSZ, LSA, [EAH,] LA***

7.5.4 DMA Get List with Fence or with Barrier Command

Like the **getl** command, the get list with fence or with barrier (**getl*<f,b>***) command transfers discontinuous blocks of data from the effective address space to a contiguous area of local storage. Unlike the **getl** command, the **getl*<f,b>*** commands provide local ordering with respect to other commands within the same tag group (that is, the same tag id). This command is not available from the MFC proxy command queue.

getlf ***CL, TG, LSZ, LSA, [EAH,] LA***
getlb ***CL, TG, LSZ, LSA, [EAH,] LA***

7.6 Put Commands (Local Storage to Main Storage)

Put commands are conventional DMA commands. Transfer size bytes of data are copied to the effective address and translated from the source local storage address by the MMU as appropriate.

When the **put** and **put*<f,b>*** commands have an “s” suffix, they can set the run bit after the DMA operation is complete. Commands with an “s” suffix can only be executed from the MFC proxy command queue.

7.6.1 DMA Put

The put (**put*[s]***) command transfers the number of bytes specified by the transfer size parameter from the local storage address of the corresponding SPU to the effective address.

put	***CL, TG, TS, LSA, [EAH,] EAL***
puts	***CL, TG, TS, LSA, [EAH,] EAL***

7.6.2 DMA Put with Fence or with Barrier

Like the **put** command, the put with fence or with barrier (**put*<f,b>[s]***) command transfers the number of bytes specified by the transfer size parameter from the local storage address of the corresponding SPU to the effective address. Unlike the **put** command, the **put*<f,b>*** commands provide local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID).

putf	***CL, TG, TS, LSA, [EAH,] EAL***
putb	***CL, TG, TS, LSA, [EAH,] EAL***
putfs	***CL, TG, TS, LSA, [EAH,] EAL***
putbs	***CL, TG, TS, LSA, [EAH,] EAL***

7.6.3 DMA Put List

The put list (**putl**) command provides software with a method to move a contiguous area of local storage to discontinuous areas in the effective address space. This command uses a list of effective address (low) or transfer size pairs (DMA list entries) stored in local storage as the destination of the DMA operation. The list address parameter of this command contains the starting address of the DMA list in local storage. The list address is a local storage address, and is therefore not translated by the MMU. The number of bytes in the list is provided in the list size parameter. The list size parameter must be a multiple of 8 bytes for this DMA command, and the list address parameter must be aligned on an 8-byte boundary in local storage. This command is not available from the MFC proxy command queue.

The DMA parameters provided in local storage must follow the same format as the **put** command. The LSA must start on a 16-byte boundary, if the transfer size is 16 bytes or less.

The effective address (high) parameter is provided as part of the **putl** command. While the starting effective address is always EAH || LEAL, a DMA transfer within a list element can cross the 232 boundary.

putl	***CL, TG, LSZ, LSA, [EAH,] LA***

7.6.4 DMA Put List with Fence or with Barrier

Like the **putl** command, the put list with fence or with barrier (**putl*<f,b>***) command transfers data from a contiguous area of local storage to discontinuous blocks in the effective address space. Unlike the **putl** command, the **putl*<f,b>*** commands provide local ordering with respect to other commands within the same tag group (that is, commands with the same tag ID). This command is not available from the MFC proxy command queue.

putlf	***CL, TG, LSZ, [EAH,] LA***
putlb	***CL, TG, LSZ*, *LSA, [EAH,] LA***

7.6.5 DMA Put Result (hint)

The **put**, **putl**, **put*<f,b>***, and **putl*<f,b>*** commands also support a "result" modifier (**putr**, **putr*<f,b>***, **putrl**, **putrl*<f,b>***). These commands perform the same operations. However, they also provide a hint to the MFC that the data for the transfer can be directly transferred into the PPE L2 cache. This is only a hint for performance. If the specified cache line is not found in the PPE L2 cache, the data will be written to system memory. This form of these commands enables the SPU to deliver results directly to the PPE by updating the PPE L2 cache.

Implementation Note:

The current implementation for the CBEA-compliant processor does not support the direct transfer of data to the PPE L2 cache (called scarfing). These commands are treated as if the "result" hint were not provided.

7.7 Storage Control Commands

The storage control commands described in this section are similar to the PowerPC storage control commands defined in *PowerPC Architecture, Book II*. SL1 storage control commands affect the transfer size bytes of data. They are either flushed from, pre-fetched into, or zeroed in the SL1 beginning from the specified effective address and translated as appropriate by the MMU. These storage control commands all have an implied tag-specific barrier. Thus, the initial storage control command and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue that have the same tag group ID. Subsequent commands in the same tag group as the SL1 storage control command will not start until the storage control command has completed.

7.7.1 SL1 Data Cache Range Touch Command

The SL1 data cache range touch (**sdcrt**) command is a hint to the MFC that the SPU will probably issue a DMA **get** operation to the range of addresses specified by the effective address and transfer size parameters. Execution of the **sdcrt** command does not cause the system error handler to be invoked. Updates of the reference and change (RC) bits within the page table entry (PTE) are not required for this operation.

sdcrt	***CL, TG, TS, [EAH,] EAL***

Note: This command has no effect on the cache lines in the atomic update facility. However, it does affect data in the SL1 cache if the cache is implemented.

7.7.2 SL1 Data Cache Range Touch for Store Command

The SL1 data cache range touch for store (**sdcrtst**) command is a hint to the MFC that the SPU will probably issue a DMA **put** operation to the range of addresses specified by the effective address and transfer size parameters. Execution of the **sdcrtst** command does not cause the system error handler to be invoked. Reference and change recording are not required for this operation.

sdcrtst ***CL, TG, TS, [EAH,] EAL***

Note: This command has no effect on the cache lines in the atomic update facility. However, it does affect data in the SL1 cache if the cache is implemented.

7.7.3 SL1 Data Cache Range Set to Zero Command

The SL1 data cache range set to zero (**sdcrz**) command sets the range of storage specified by the effective address or list address and transfer size or list size parameters to zero. This command does not cause the data to exist in the SL1 if the storage area is caching inhibited.

sdcrz ***CL, TG, TS, [EAH,] EAL***

Note: All the bytes in the block containing the byte addressed by the effective address are set to ‘0’. This command can affect data cached in the atomic update facility.

Implementation Note:

However, this command does not cause the data to be brought into the atomic update facility cache if the cache line was not present before the execution of the command.

7.7.4 SL1 Data Cache Range Store Command

If an effective address in the range defined by the effective address/list address and transfer/list size parameters is in memory coherence required storage, and a data block in any data cache of the system is considered modified, the SL1 data cache range store (**sdcrst**) command writes the modified blocks to main storage. The data blocks can remain in the cache, but are no longer considered modified.

If the storage is not memory coherency required and a data block in the SL1 of the issuing SPU is considered modified, the modified blocks are written to main storage. The data blocks can remain in the cache, but are no longer considered modified.

Updates of R and C bits of the PTE are not required for this operation.

sdcrst ***CL, TG, TS, [EAH,] EAL***

Implementation Note:

This command affects data cached in the atomic update facility caches if a block containing a byte addressed by the effective address and any locations in the block are considered modified.

7.7.5 SL1 Data Cache Range Flush Command

If an effective address in the range defined by the effective address, or list address and transfer, or list size parameters is in memory coherency required storage, and a data block in any data cache of the system is considered modified, the modified blocks are written to main storage. All data blocks in the effective address range are invalidated in the data caches of all processors.

If the storage is not memory coherency required, and a data block in the SL1 of the issuing SPU is considered modified, the SL1 data cache range flush (**sdcrf**) command writes the modified blocks to main storage. All data blocks in the effective address range are invalidated in the data cache of the issuing SPU.

Updates of R and C bits of the PTE are not required for this operation.

sdcrf ***CL, TG, TS, [EAH,] EAL***

Implementation Note:

This command affects data cached in the atomic update facility caches, if a block containing a byte addressed by the effective address and any locations in the block are considered modified. If the block is present in the atomic update facility cache, it is invalidated.

7.8 MFC Atomic Update Commands

There are four MFC atomic update commands: **getlar**, **putllc**, **putlluc**, **putqlluc.** The **getlar**, **putllc**, and **putlluc** MFC atomic update commands are performed without waiting for other commands in the MFC SPU queue and thus have no associated tag. The MFC performs the atomic update operations independently of any pending **mfcsync**, **mfceieio**, or **barrier** commands in the MFC SPU command queue. Software must issue a read (**rdch**) from the MFC Read Atomic Command Status Channel (see page 120) after issuing each atomic update command to verify completion of the command.

The MFC commands for get lock line and reserve (**getllar**) and put lock line conditional (**putllc**) provide similar functionality as the PowerPC **lwarx**, **ldarx**, **stwcx**, and **stdcx** instructions for use in atomic updates. The put lock line unconditional (**putlluc**) command and the put queued lock line unconditional (**putqlluc**) command perform a similar function to a cacheable store instruction in the PowerPC Architecture, conventionally used by software to release a “lock”. The difference between the **putlluc** and **putqlluc** commands is that the **putqlluc** command is tagged and queued behind other MFC commands in the MFC SPU command queue, whereas the **putlluc** command is executed immediately. Since the **putqlluc** command is tagged and has an implied tag-specific fence, it is then ordered with respect to all other commands in the same tag group all ready in the MFC SPU command queue.

The **getllar**, **putllc**, and **putlluc** commands are not tagged; therefore, they are executed immediately. Even though the **getllar**, **putllc**, and **putlluc** commands are executed immediately, these commands still require an available slot in the MFC SPU command queue. No ordering with other commands in the MFC SPU command queue should be assumed. After issuing each **getllar**, **putllc**, or **putlluc** command, the software must issue a read from the MFC Read Atomic Command Status Channel (see page 120) to verify completion of the command. MFC atomic update commands must be issued to memory coherent and cacheable pages. Issuing a **getllar**, **putllc**, **putlluc**, or **putqlluc** command to non-memory coherent or non-cacheable pages is undefined. See the *PowerPC Architecture, Book II* for examples of the use of atomic updates. The MFC atomic update commands can only be issued to the MFC SPU command queue by following the sequence described in *Section 9.2 MFC SPU Command Issue Sequence* beginning on page 110.

7.8.1 Get Lock Line and Reserve Command

A get lock line and reserve (**getllar**) command is similar to the operation of the PowerPC **lwarx** and **ldarx** instructions with the exception of the size of the data transfer. The data transfer size of the **getllar** command is a cache line.

Issuing the **getllar** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, it executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command has completed results in an error (MFC command queue processing is halted, and the PPE is interrupted).

The **getllar** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

A read channel (**rdch**) from the MFC Read Atomic Command Status Channel (see page 120) must be performed after issuing this command and before issuing another **getllar**, **putllc,** or **putlluc** command.

Privileged software is required to issue a **putllc** command to a privileged address to reset the reservation of an application as part of the SPE context switch. Issuing a **putllc** command resets the reservation, if a context switch occurred after the **getllar** made a reservation, but before the **putllc** used the reservation. This prevents the resumed context from inadvertently using the reservation of the previous context if it was switched at a similar point using the same address.

Software must avoid issuing the **getllar** command successively with the same effective address (having the same reservation granule) without an intervening **putllc** command unless a random backoff technique is used to avoid livelock situations. A random backoff technique provides a random amount of delay between issues of two or more successive **getllar** commands. A livelock (an endless loop in program execution) can occur during locking or **barrier** sequences, such as test and set, compare and swap, or repeatedly polling for a value to change, using the **getllar** command, before performing an atomic update or other action.

When using the atomic update sequence in a **barrier** or synchronization operation (such as compare and swap or test and set), consider using the Lock Line Reservation Lost Event (see page 153) instead of repeatedly issuing the **getllar** command. Using this event allows the program to accomplish other tasks while waiting for an external modification to the lock line data. If no other task is available, the programmer is able to perform a read channel (**rdch**) from the SPU Read Event Status Channel (see page 136) to put the SPU into low power state until the lock line data has been modified.

Note: When a change is made to the lock line that is being monitored by issuing another **getllar** to a different effective address, a reservation lost event for the previous lock line can be received. In addition a reservation lost event can also be received due to an context switch. This event only specifies that data in the cache line might have been modified. Both of these cause an additional cycle through the **getllar**, compare and test, and wait loop.

7.8.2 Put Lock Line Conditional Command

A put lock line conditional (**putllc**) command is similar to the operation of the PowerPC **stwcx.** and **stdcx.** instructions with the exception of the size of the store conditional. The store conditional size of the **putllc** command is a cache line.

Issuing the **putllc** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, this command executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command has completed results in an error (MFC command queue processing is halted, and the PPE is interrupted).

The **putllc** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

The **putllc** command is a conditional store operation. The store is not successful if no reservation for the same address has been made, or if the reservation has been lost. A read of the MFC Read Atomic Command Status Channel (see page 120) is required to determine the completion with success, or with failure of this command. An SPU indefinite stall will result if the MFC Read Atomic Command Status Channel (see page 120) is read without the issuance of a lock line MFC command.

7.8.3 Put Lock Line Unconditional Command

A put lock line unconditional (**putlluc**) command is similar to the operation of the **putllc** command, but the store for the **putlluc** is always performed. The **putlluc** command store is not dependent upon the existence of a previously made reservation. The store size of the **putlluc** command is a cache line.

If an attempt is made to issue this command before a previous **getllar**, **putllc**, or **putlluc** command completes, an error results, the MFC command queue processing is halted, and the PPE is interrupted.

The **putlluc** ***CL, LSA, [EAH,] EAL*** command cannot be issued from the MFC proxy command queue.

Issuing the **putlluc** command requires an available MFC SPU command queue slot. However, this command is issued immediately, is not queued behind other commands, and has no associated tag. Therefore, it executes independently of pending **mfcsync**, **mfceieio**, or **barrier** commands in the queue. An attempt to issue this command before a previous **getllar**, **putllc**, or **putlluc** command completes results in an error (MFC command queue processing is halted, and the PPE is interrupted).

This command is a store operation. The store is not conditional on having acquired a reservation. A read of the MFC Read Atomic Command Status Channel (see page 120) is required to clear the status of the operation. A read of the MFC Read Atomic Command Status Channel is also required to verify the completion of this command. An SPU indefinite stall will result if the MFC Read Atomic Command Status Channel is read without the issuance of a put lock line DMA command.

7.8.4 Put Queued Lock Line Unconditional Command

The put queued lock line unconditional (**putqlluc**) command is functionally equivalent to the put lock line unconditional (**putlluc**) command. The difference between the two commands is the order in which the commands are performed and how completion is determined. The **putlluc** command is performed immediately and the **putqlluc** command is placed into the MFC SPU command queue along with other MFC commands.

No ordering is performed between the queued **putqlluc** command and the immediate **getllar**, **putllc**, or **putlluc** commands. Thus, a **putqlluc** can execute before, or after, a subsequently-issued, or a previously-issued immediate atomic command (**getllar**, **putllc**, or **putlluc**); that is, do not assume any order of execution with respect to immediate lock-line commands.

Since this command is queued, it executes independently of any pending immediate **getllar**, **putllc**, or **putlluc** commands. To determine if the **putqlluc** command is complete, software must wait for a tag-group completion. See *Section 9.3 MFC Tag-Group Status Channels* beginning on page 111, for more details.

The **putqlluc** command contains a tag parameter and has an implied tag-specific fence. The implied tag-specific fence prevents this command from being issued until all previously issued commands with the same tag have completed. Unlike the immediate form of this command, multiple **putqlluc** commands can be issued and pending in the DMA command queue. All the MFC command queue ordering rules apply to the **putqlluc** command.

The **putqlluc** ***CL, TG, LSA, [EAH,] EAL*** command cannot be issued from MFC proxy command queue.

This command is a store operation. The store is not conditional on having acquired a reservation.

Programming Note:

The put queued lock line unconditional (**putqlluc**) command allows software to queue the release of a lock behind the commands accessing storage associated with the lock. For proper operation, the **putqlluc** command must be within the same tag group as the commands accessing the associated storage or other ordering commands must be used. In addition, either **mfceieio** or **mfcsync** commands must also be used (as appropriate).

7.9 MFC Synchronization Commands

MFC synchronization commands are used to control the order in which storage accesses are performed with respect to other MFCs, processors, and other devices.

Many commands support the embedded tag-specific fence-form modifier, or barrier-form modifier. The notation <***f,b***> indicates that either the tag-specific fence or tag-specific barrier form is available. The tag-specific fence, **<f>**, ensures that this command is ordered with respect to all preceding commands in the DMA command queue within the same tag group. Any subsequent command with the same tag ID that is not a fence-form or barrier-form, or any command within a different tag group, can be performed out-of-order with respect to this command or previously-issued commands. The tag-specific barrier, <***b***>, ensures that this command and all subsequent commands within the same tag group as this command are ordered with respect to all preceding commands in the DMA command queue within the same tag group. This command and all subsequent commands with the same tag ID as this command will not be performed out-of-order with respect to all preceding commands with the same tag ID as this command. Once all previously-issued commands with the same tag ID as this command have been performed, this command and subsequent commands can be performed. The order in which these subsequent commands are performed is determined by subsequent fence and barrier form commands. Commands with a fence or barrier are not ordered with respect to subsequent commands.

The **fence** and **barrier** modifiers provide stronger consistency of storage accesses in the weakly consistent storage model of the CBEA for several combinations of storage accesses involving commands in the same tag group. This allows programmers to avoid the requirement to add additional synchronization commands for these specific combinations if the commands are in the same tag group and either the **fence** or **barrier** modifier is used. The following DMA combinations require no additional synchronization commands to provide an ordering function when both commands access storage having the same storage attributes.

- Same tag group **put(l)** type or **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or the barrier **<b>** modifier.

- Same tag group **get(l)** type command followed by a **get(l)** type command either the fence **<f>** or the barrier **<b>** modifier.
- Same tag group **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or the barrier **<b>** modifier.

If the storage accesses between two DMA operations that access storage having different storage attributes need to be strongly ordered, use of either the fence **<f>** or barrier **<b>** modifier is required, but is insufficient to guarantee the ordering function with respect to other units or processors. In these cases, an **mfcsync** command MUST be issued between commands involving storage with different storage attributes to provide the required ordering function.

- A same tag group **put(l)** command or **sdcrz** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are memory coherence required and are neither write through required nor caching inhibited must be separated by an **mfcsync** command to provide the required ordering function.
- A same tag group **put(l)** command followed by a **get(l)** command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and guarded must be separated by an **mfceieio** command to provide the required ordering function.

Implementation Note:

The CBEA specifies that the fence and barrier modifiers provide stronger storage access consistency for some combinations of DMA commands. The following MUST be observed by any CBEA-compliant hardware implementation:

- A **put(l)** type or **sdcrz** command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are memory coherence required and is neither write through required nor caching inhibited has a storage order effect at least as strong as two stores on the PPE separated by a **lwsync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are memory coherence required and is neither write through required nor caching inhibited has a storage order effect at least as strong as two loads on the PPE separated by a **lwsync** instruction.
- A **put(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited has a storage order effect at least as strong as two stores on the PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and not guarded has a storage order effect at least as strong as two loads on the PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and guarded has a storage order effect at least as strong as two loads on the PPE separated by an **eieio** instruction.
- A **put(l)** type command followed by a **get(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and not guarded has a storage order effect at least as strong as a store followed by a load on the PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are memory coherence required and is neither write through required nor caching inhibited has a storage order effect at least as strong as a load followed by a store on the PPE separated by a **lwsync** instruction.

- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and not guarded has a storage order effect at least as strong as a load followed by a store on the PPE separated by a **sync** instruction.
- A **get(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and guarded has a storage order effect at least as strong as a load followed by a store on the PPE separated by an **eieio** instruction.
- A **put(l)** type command followed by a **put(l)** type command with either the fence **<f>** or barrier **<b>** modifier where the storage attributes are caching inhibited and guarded are always performed in program order with respect to any processor or mechanism.

An example of the use of a fence-form modifier is when multiple MFC DMA commands are needed to load an SPU program and to start its execution. In this example, one MFC DMA command is used to load the data segment and the second MFC DMA command with both the SPU start ‘‘s” and the tag-specific fence <**f**> modifiers is used to load text data. As long as the two commands have the same tag ID, the fence ensures that the load of the data segment is completed before loading the text data and before starting the SPU program execution. Without the fence, the second MFC DMA command could complete and could start the SPU program before the data segment is loaded.

An example of the use of a barrier-form modifier is when a read of a buffer takes multiple commands and must be performed before writing the buffer, which also takes multiple commands. In this example, the commands to read the buffer can be queued and performed in any order. Using the barrier-form for the first command to write the buffer allows the commands used to write the buffer to be queued without waiting for the MFC Tag-Group Status Update Event on the read commands. (The barrier-form is only required for the first buffer write command.) As long as the buffer read and buffer write commands have the same tag ID, the barrier ensures that the buffer is not written before being read. If multiple commands are used to read and write the buffer, using the barrier option allows the read commands to be performed in any order and the write commands to be performed in any order, which provides better performance, but forces all reads to finish before the writes start.

7.9.1 MFC Synchronize Command

An MFC synchronize (**mfcsync**) command is similar in operation to the PowerPC **sync** instruction.

mfcsync ***TG***

This command is used to control the order in which DMA commands within this tag group are executed with respect to other processors and devices. The **mfcsync** command implies a tag-specific barrier. This means that the **mfcsync** command and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue with the same tag ID. A storage area with the same storage attributes (that is, memory coherence required and neither write through required nor caching inhibited) is required when a storage access ordering function is needed between a put(l) command and a get(l) command. This storage area (with the same storage attributes) is also required when an ordering function is needed between any two commands, which have dissimilar storage attributes.

Programming Note:

To obtain the best performance across the widest range of implementations, the programmer should use either the **barrier** command, fence **<f>** or barrier **<b>** options, or the **mfceieio** command if any of these are sufficient for the ordering needs.

7.9.2 MFC Enforce In Order Execution of I/O Command

An MFC enforce in-order execution of I/O (**mfceieio**) command is similar in operation to the PowerPC **eieio** instruction.

mfceieio ***TG***

This command is required when a storage access ordering function is needed between a **put(l)** type command and a **get(l)** command involving storage having the same storage attributes of caching inhibited and guarded.

The **mfceieio** command implies a tag-specific barrier. This means that the **mfceieio** command and all subsequent commands in the same tag group are ordered with respect to all previous commands in the queue with the same tag ID.

Programming Note:

This command is intended for use in managing shared data structures, in performing memory-mapped I/O, and in preventing load and store combining in system memory. In order to obtain the best performance across the widest range of implementations, the programmer should use either a fence or barrier form of a command, or if neither of these is sufficient for ordering, the **mfceieio** command.

7.9.3 Barrier Command

A **barrier** command orders all subsequent commands with respect to all commands preceding the **barrier** command in the DMA command queue, independent of tag group IDs. The **get*<f,b>***, **getl*<f,b>***, **put*<f,b>***, and **putl*<f,b>*** commands have an embedded form of fence or barrier that only affects commands within the same tag group.

barrier ***TG***

The **barrier** command is not tag-specific. The specified tag can be used to determine when the command is complete. The **barrier** command does not complete until all preceding commands in the queue are complete. Subsequent commands in the queue begin when the **barrier** command completes.

The barrier command has the same storage access ordering properties independent of the tag group as does the use of the embedded **fence** or **barrier** modifier for commands in the same tag group.

7.9.4 Send Signal Command

A send signal (**sndsig**) command logically sets signal bits in the targeted signal notification register. The targeted signal notification register is specified by the effective address. The data used to update the signal notification register is the data from the location specified by the local storage address.

This operation is always a 4-byte operation, and the transfer size (***TS***) parameter must be set to four.

sndsig ***CL, TG, TS, LSA, [EAH,] EAL***

The **sndsig** command is normally used to signal events between SPUs, between the PPE and SPUs, or between SPUs and I/O devices. Each SPU has two signal notification registers that can be targeted by this command. The PPE has no signal control words, and it can only initiate signals in this manner. I/O devices can initiate signals in this manner and optionally, I/O devices can also have signal control words.

The signal notification registers can be programmed for overwrite mode or logical OR mode. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 221. In overwrite mode, the contents of the signal notification register are replaced with the signal information set by this command. This mode is useful in a one-to-one signalling environment. In the logical OR mode, the contents of the signal notification register are logically ORed with the signal information set by this command. This mode is useful in a many-to-one signalling environment.

In logical OR mode, the contents of the signal control word are logically ORed with the data written. In overwrite mode, the signal control word contents are replaced with the data written. A processor that overwrites the signal notification register data does not have hardware protection. When the signal control word is read locally by the signal control owner, any signal bits that are set are reset. Any remote (non-owner) read of these signal control words returns the current state of the signal control word, but does not result in a reset. SPUs read and reset signal control words through SPU channels.

MMIO addresses are provided for the PPE or I/O devices to issue the **sndsig** command to a specified signal notification register. The PPE can also set or clear the signal control words of SPUs to establish the context through the SPU Channel Access Facility.

An SPU that receives a **sndsig** operation must guarantee that all store operations sent to the local storage from a single source are complete before depending on the data associated with the **sndsig** operation. The MFC Multisource Synchronization Facility (see page 96) must be used to ensure that stores from multiple sources are complete.

Implementation Note:

The **sndsig** command can be implemented as a **put** command.

7.9.5 Send Signal with Fence or with Barrier Command

The send signal with fence or with barrier (**sndsig<f,b>**) command has a function similar to the **sndsig** command. However, the **sndsig*<f,b>*** commands provide local ordering with respect to other commands within the same tag group (same tag ID).

sndsigf ***CL, TG, TS, LSA, [EAH,] EAL***

sndsigb ***CL, TG, TS, LSA, [EAH,] EAL***

Implementation Note:

The **sndsig*<f,b>*** command can be implemented as a **put*<f,b>*** command.

8. Problem State Memory-Mapped Registers

The problem state memory-mapped registers define the set of facilities that an application running in problem state (that is, MSR[PR] = '1' in the PPE or MFC_SR1[PR] = '1' for an SPE) can access. Each SPE has a complete set of the registers as described in this section. The access privileges for each SPE is independent since the registers are allocated in different real address pages. Access to these registers is controlled by privileged software when mapping the corresponding address into the effective address of an application in the page table.

If an SPE is used as a privileged resource (that is, when MFC_SR1[PR] = '0'), access to these registers should also be marked as privileged. For more information on access privileges, see *PowerPC Architecture, Book III*.

The CBEA is flexible enough that the problem state memory-mapped registers can be managed by the PPE, by other processors, or by other SPUs.

Table 8-1. SPE Problem State Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Multiisource Synchronization Area			
x‘00000’	MFC_MSSync	MFC Multisource Synchronization Register (see page 96)	Read/Write
MFC Command Parameter Area			
x‘03000’	Reserved	Reserved for future expansion.	Reserved
x‘03004’	MFC_LSA	MFC Local Storage Address Register (see page 75)[1]	Write Only
x‘03008’	MFC_EAH	MFC Effective Address High Register (see page 76)[1]	Write Only
x‘0300C’	MFC_EAL	MFC Effective Address Low Register (see page 77)[1]	Write Only
x‘03010’	MFC_Size	MFC Transfer Size Register (see page 74)[1, 3] (Upper 16 bits of register).	Write Only
	MFC_Tag	MFC Command Tag Register (see page 73)[1, 3] (Lower 16 bits of register)	Write Only
x‘03014’	MFC_ClassID	MFC Class ID Register (see page 72)[1, 2] (Upper 16 bits of register for write)	Write Only
	MFC_CMD	MFC Command Opcode Register (see page 71) [1, 3] (Lower 16 bits of register for write)	Write Only
	MFC_CMDStatus	MFC Command Status Register (see page 80)(all 32 bits for read)	Read Only
MFC Command Queue Control Area			
x‘03020’:x‘030FF’	Reserved	Reserved	
x‘03104’	MFC_QStatus	MFC Queue Status Register (see page 81)	Read Only
x‘03204’	Prxy_QueryType	Proxy Tag-Group Query Type Register (see page 83)	Read/Write
x‘0321C’	Prxy_QueryMask	Proxy Tag-Group Query Mask Register (see page 84)	Read/Write
x‘0322C’	Prxy_TagStatus	Proxy Tag-Group Status Register (see page 85)	Read Only
x‘03330’:x‘03FFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_ClassID and MFC_CMD registers must be written with a single 32-bit store instruction.
3. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.

Table 8-1. SPE Problem State Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
SPU Control Area			
x‘04004’	SPU_Out_Mbox	SPU Outbound Mailbox Register (see page 91)	Read Only
x‘0400C’	SPU_In_Mbox	SPU Inbound Mailbox Register (see page 92)[1]	Write Only
x‘04014’	SPU_Mbox_Stat	SPU Mailbox Status Register (see page 93)	Read Only
x‘0401C’	SPU_RunCntl	SPU Run Control Register (see page 86).	Read/Write
x‘04024’	SPU_Status	SPU Status Register (see page 87).	Read Only
x‘04034’	SPU_NPC	SPU Next Program Counter Register (see page 89)	Read/Write
x‘04038’:x‘13FFF’	Reserved	Reserved	
Signal-Notification Area			
x‘1400C’	SPU_Sig_Notify_1	SPU Signal Notification 1 Register (see page 94)	Read/Write
x‘14010’:x‘1BFFF’	Reserved	Reserved	
x‘1C00C’	SPU_Sig_Notify_2	SPU Signal Notification 2 Register (see page 95)	Read/Write
x‘1C010’:x‘1FFFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_ClassID and MFC_CMD registers must be written with a single 32-bit store instruction.
3. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.

8.1 MFC Command Parameter Registers

These subsections provide details on the MFC Command Parameter Registers. These registers are:

- *Section 8.1.1 MFC Command Opcode Register* beginning on page 71
- *Section 8.1.2 MFC Class ID Register* beginning on page 72
- *Section 8.1.3 MFC Command Tag Register* beginning on page 73
- *Section 8.1.4 MFC Transfer Size Register* beginning on page 74
- *Section 8.1.5 MFC Local Storage Address Register* beginning on page 75
- *Section 8.1.7 MFC Effective Address Low Register* beginning on page 77
- *Section 8.1.6 MFC Effective Address High Register* beginning on page 76

8.1.1 MFC Command Opcode Register (MFC_CMD)

The MFC Command Opcode Register determines the operation to be performed. The validity of the opcode is checked asynchronous to the instruction stream. If the MFC command or any of its parameters are invalid, MFC SPU command queue processing is suspended and, if enabled, an invalid MFC command interrupt is sent. For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

The MFC Command Opcode must be written to the MFC proxy command queue along with the MFC Class ID parameters using a single 32-bit store instruction. The MFC Command Opcode is the lower 16 bits of the 32-bit word. The upper 8 bits of the Command Opcode field are reserved. If the MSb (that is, bit 0 of this field) is set to ‘1’, the command is reserved and can be used for an implementation-dependent function.

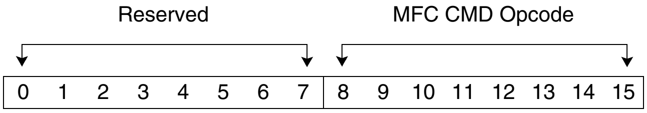
Software must avoid programming practices that enqueue commands with forward dependencies on commands that are enqueued later. Software with this type of dependency can create a deadlock, because of the number of available slots in the MFC command queues. In addition, while queue depth is implementation dependent, software must not be written to require a specific queue depth.

Software can determine the number of queue slots available in the MFC proxy command queue by reading the MFC Queue Status Register (see page 81). The value returned is the number of available queue slots. Software can use this value to avoid repeating the queuing sequence for a full MFC proxy command queue.

The queuing sequence for MFC proxy commands is described in *Section 8.2 MFC Proxy Command Issue Sequence* on page 78.

Access Type MMIO: Read[1]/Write

Base Address Offset (BP_Base | PS(n)) + x‘03014’; where n is the SPE number (lower 16 bits)



Bits	Field Name	Description
0:7	Reserved	Should be set to zeros. If bit 0 is set to ‘1’, the opcode is reserved.
8:15	MFC CMD Opcode	MFC command opcode.

Programming Note:

The total number of queue slots is implementation dependent and varies between implementations. For portability of an application, the queuing sequence for MFC commands and the method to determine the number of queue slots available should be provided as a macro or as a library function that uses a configuration-dependent method for queuing commands.

[1] When read, this register returns the value in the MFC Command Status Register (see page 80).

8.1.2 MFC Class ID Register (MFC_ClassID)

The MFC Class ID Register is used to specify the Replacement Class ID and the Transfer Class ID for each MFC command. These IDs are used by the processor and any software to improve the overall performance of the system. The exact function and mapping of the class IDs is implementation-dependent. For more information, see *Section 19 Cache Replacement Management Facility* beginning on page 231.

The Transfer Class ID (TclassID) part of this register is used to identify access to storage with differing characteristics. The TclassID is intended to be used to allow an implementation to optimize the transfers corresponding to the MFC command based on the characteristics of the storage location. Setup and use of the TclassID is implementation dependent.

The contents of the MFC Class ID Register are not persistent and must be written for each MFC command queuing sequence.

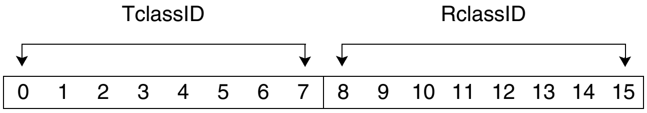
The MFC Class ID Register performs the same function, whether it is used with commands issued from the MFC proxy command queue or the MFC SPU command queue. This register is used to control resources associated with an SPU and it has no effect on resources associated with the PPE.

The validity of the MFC Class ID Register is not verified. The number of class IDs supported is implementation dependent. The default class ID (x‘00’) is used for all undefined or invalid class IDs.

An invalid class ID does not generate an interrupt.

Access Type MMIO: Read[1]/Write

Base Address Offset (BP_Base | PS(n)) + x‘03014’; where n is the SPE number (upper 16 bits)



Bits	Field Name	Description
0:7	TclassID	Transfer class identifier.
8:15	RclassID	Replacement class identifier.

Note: The MFC Class ID Register (see page 72) must be written to the MFC proxy command queue along with the MFC Command Opcode Register (see page 71) using a single 32-bit store instruction. The MFC proxy command parameter is the lower 16 bits of a 32-bit word; the MFC Class ID parameter is the upper 16 bits of the 32-bit word.

[1] When read, this register returns value in the MFC Command Status Register (see page 80).

8.1.3 MFC Command Tag Register (MFC_Tag)

The MFC Command Tag Register is used to specify an identifier for each command or group of commands. The tag can be any value between x‘0’ and x‘1F’. Tags have a purely local scope in the hardware.

Any number of MFC commands can be tagged with the same ID. MFC commands tagged with the same ID are referred to as a tag group. Tags on commands in the MFC proxy command queue are independent from tags on commands in the MFC SPU command queue.

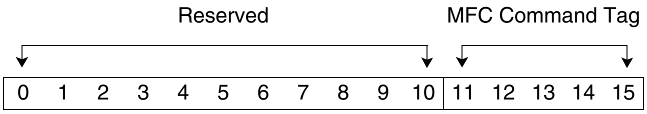
The contents of the MFC Command Tag Register are not persistent and must be written for each MFC-command queuing sequence.

The validity of this register is checked asynchronous to the instruction stream. If the upper bits (bits ‘0’ through ‘10’) are not set to ‘0’, the command queue processing is suspended and, if enabled, an invalid MFC command interrupt is sent. For more information, see *Section 21 Interrupt Facilities* beginning on page 237.

Note: Software is required to write the MFC Command Tag Register for each command, even if the command does not support this parameter. Even though it is written, the tag is not used. For future compatibility, software should set the MFC Command Tag to zero for all commands that do not support the MFC command tag.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘03010’; where n is the SPE number (lower 16-bit)



Bits	Field Name	Description
0:10	Reserved	Set to zeros.
11:15	MFC Command Tag	MFC command tag.

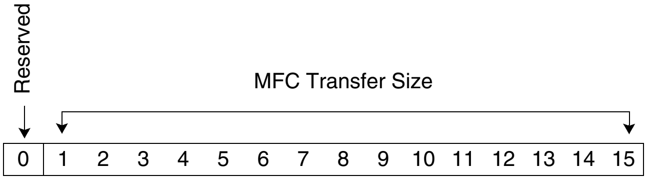
Note: The MFC Command Tag Register must be written along with the MFC Transfer Size Register (see page 74) using a single 32-bit store instruction to the MFC proxy command queue. The MFC Command Tag parameter is the lower 16 bits of the 32-bit value written to the base address offset.

[1] An implementation should support Read access to this facility for diagnostic purposes.

8.1.4 MFC Transfer Size Register (MFC _Size)

The MFC Transfer Size Register is used to specify the size of an MFC transfer. The size can have a value of 1, 2, 4, 8, 16, or a multiple of 16 bytes to a maximum of 16 KB. The contents of the MFC Transfer Size Register are not persistent and must be written for each MFC SPU command enqueue sequence.
The validity of this register is checked asynchronous to the instruction stream. If the size is invalid, the MFC SPU command queue processing is suspended, and, if enabled, an MFC DMA alignment interrupt is sent. For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Access Type	MMIO: Write Only[1]
Base Address Offset	(BP_Base I PS(n)) + x‘03010’; where n is the SPE number (upper 16 bits)



Bits	Field Name	Description
0	Reserved	Set to zeros.
1:15	MFC Transfer Size	MFC transfer size. Allowable MFC transfer sizes are: • 1, 2, 4, and 8 bytes (naturally aligned), where the source and destination address *must* have the same 4 least-significant bits (see *PowerPC Architecture, Book II* for more information on MFC alignment). • 16 bytes and multiples of 16 bytes up to 16 KB, where the source and destination addresses must be 16-byte (quadword) aligned.

Note: The MFC Transfer Size Register must be written along with the MFC Command Tag Register using a single 32-bit store instruction for the MFC proxy command queue. The MFC Transfer Size Register is the upper 16 bits of the 32-bit value written to the base address offset.

Programming Note:

DMA transfers of less than one cache line should be used sparingly. Excessive use of small transfers can result in poor bus and memory bandwidth use. Due to their low performance and high setup cost, transfers of less than 16 bytes should only be used when necessary to interface with an I/O device. When transferring 128 bytes or more, programmers should attempt to align the source and destination on 128-byte boundaries for optimal performance.

[1] An implementation should support Read access to this facility for diagnostic purposes.

8.1.5 MFC Local Storage Address Register (MFC_LSA)

The MFC Local Storage Address Register is used to supply the SPU local storage address associated with an MFC command to be queued. This address is used as the source or destination of the MFC transfer as defined in the MFC command. For more information, see *Section 7 MFC Commands* beginning on page 47.

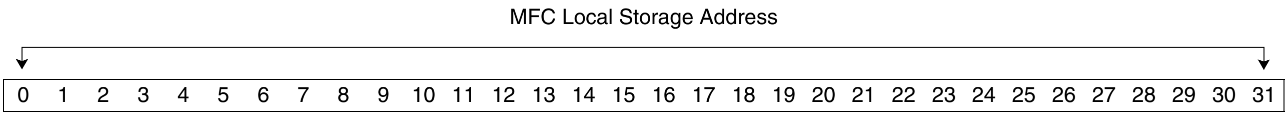
The contents of the MFC Local Storage Address Register are not persistent and must be written for each MFC command queuing sequence.

The validity of this register is checked asynchronous to the instruction stream. To be considered aligned, the four least significant bits of the local storage address must match the least-significant four bits of the effective address.

If the address is unaligned, the processing for both command queues is suspended and an MFC DMA alignment interrupt is sent. For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘03004’; where n is the SPE number



Bits	Field Name	Description
0:31	MFC Local Storage Address	MFC local storage address. **Note:** The 4 least-significant bits of the local storage address must match the least-significant four bits of the effective address.

Programming Note:

For optimum performance, transfers of greater than 16 bytes should have bits[25:31] of the local storage address set to 0.

[1] An implementation should support Read access to this facility for diagnostic purposes.

8.1.6 MFC Effective Address High Register (MFC_EAH)

The MFC Effective Address High Register is used to specify the effective address for the MFC command. If translation is enabled in MFC State Register One (see page 197) (that is, MFC_SR1[R] = ‘1’), effective addresses are translated into real addresses by the address-translation mechanism described in *PowerPC Architecture, Book III*.

The contents of the MFC effective address (high) register are not persistent. If not written for each command the value for EAH is automatically set to ' 0'. The high-order 32 bits of the MFC Effective Address High Register are optional. If not written, the high address bits are set to zero (that is, the address is between 0 and 4 GB).

The validity of this parameter is checked asynchronous to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, or a protection violation), MFC SPU command queue processing is suspended and an interrupt is sent.

The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data segment interrupt is sent.
- If a mapping fault or page protection violation occurs, an MFC data-storage interrupt is sent.

For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Note: The validity of the effective address can be checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type	MMIO: Write Only[1]
Base Address Offset	(BP_Base I PS(n)) + x‘03008’; where n is the SPE number

High Word of 64-bit Effective Address (Optional)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	High Word of 64-bit Effective Address (Optional)	High word of the 64-bit effective address. Zeros are used for the upper 32 bits of the effective address if this register is omitted from the command. If translation is disabled (the effective address equals the real address), some higher-order bits must also be zero, depending on the amount of real memory in the system.

[1] An implementation should support Read access to this facility for diagnostic purposes.

8.1.7 MFC Effective Address Low Register (MFC_EAL)

The MFC Effective Address Low Register is used to specify either the effective address for the MFC command. If translation is enabled in the MFC State Register One (see page 197) (that is, MFC_SR1[R] = ‘1’), effective addresses are translated into real addresses by the address-translation facility described in *PowerPC Architecture, Book III*.

The contents of the MFC Effective Address Low Register are not persistent and must be written for each MFC command enqueue sequence.

For transfer sizes less than 16 bytes, MFC_EAL bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be zeros.

If translation is disabled:

- MFC_EAL must be within the real address space limits of main storage, and the
- MFC_EAL must be a supported address within the real address space of main storage domain. Handling of unsupported addresses is implementation-dependent. For more information on the real address space of an implementation, refer to the specific implementation documentation.

In addition to these limitations, bits 28 through 31 must match bits 28 through 31 of the MFC_LSA. If any of these conditions are not true, the MFC_EAL parameter is invalid and considered unaligned.

The validity of this parameter is checked asynchronous to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, a page protection violation, or an unaligned address), MFC SPU command queue processing is suspended and an interrupt is sent.

The following types of interrupts can be sent:

- If a segment fault occurs, an MFC data segment interrupt is sent.
- If a mapping fault or page protection violation occurs, an MFC data-storage interrupt is sent.
- If the address is not aligned, a DMA alignment interrupt is sent.

For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Note: The validity of the effective address can be checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘0300C’; where n is the SPE number

Low Word of 64-bit Effective Address

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	Low Word of 64-bit Effective Address	Low word of the 64-bit effective address. For transfer sizes less than 16 bytes, address bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be zeros. For optimal performance of transfers of 128 bytes or more, the source and destination transfer addresses should be 128-byte aligned (bits 25 through 31 set to zero). If translation is disabled (the effective address equals the real address), some higher-order bits must also be zero, depending on the amount of real memory in the system.

8.2 MFC Proxy Command Issue Sequence

The CBEA requires software to follow a particular sequence to enqueue an MFC command. If the correct sequence is not followed, an MFC command is not queued and an invalid command sequence is posted in the MFC Command Status Register (see page 80) (that is, MFC_CMDStatus[RC] = '01' or '11').

To enqueue an MFC command from the PPE, the MFC parameters must first be written to the MFC command address space in the following sequence:

1. Set the local storage address.
2. Set the MFC effective address.
3. Set the MFC size and the MFC tag.1
4. Set the transfer class and replacement management class IDs and the MFC command.[2]
5. Read the MFC command status.[3]
6. Read of the MFC command status causes an enqueuing attempt of specified command and its parameters.
7. Restart the MFC command issue sequence, starting at step 1, if the status indicates failure, or there is no slot available in the MFC proxy command queue.

These parameters are held in the corresponding registers. The read of the MFC command status causes the data held in these registers to be enqueued, if the sequence has not been interrupted and if there is a slot in the MFC proxy command queue.

Implementation Note:

The hardware should set an internal state bit indicating that a command is pending for the queue when the MFC Local Storage Address Register is written. The CMD_Pending bit should be reset if the next operation to the MFC queue space does not follow the sequence outlined above. An MFC command fails if the

[1] An implementation should support Read access to this facility for diagnostic purposes.

[2] The MFC Transfer Size Register (see page 74) and the MFC Command Tag Register (see page 73), pair of registers, and the MFC Class ID Register (see page 72) and the MFC Command Tag Register (see page 73) pair must be updated using a single store instruction.

[3] Reading the MFC Command Status Register (see page 80) causes the parameters to be enqueued if the sequence is correct.

CMD_Pending bit is reset when the MFC Command Status Register is read. An MFC command can also fail due to insufficient room in the MFC command queue. The CMD_Pending bit should also be reset when a load of the MFC Command Status Register is made. The MFC command must not be put into the command queue until a load of the MFC Command Status Register is executed by the PPE and the CMD_Pending bit is set.

8.3 MFC Proxy Command Queue Control Registers

The MFC provides an MFC SPU command queue, as well as a separate MFC proxy command queue for the PPE. The commands and procedures for using the MFC SPU Command queue are defined in *Section 9 Synergistic Processor Unit Channels* beginning on page 99. The registers defined in this section are used to control the MFC proxy command queue for the PPE.

The MFC Command Parameter Registers (see page 70) along with the MFC Command Status Register (see page 80) are used for queuing an MFC command to the MFC proxy command queue for the PPE.

The procedure for queuing a command is outlined in *Section 8.2 MFC Proxy Command Issue Sequence* beginning on page 78. These registers are intended to be used by software running on the PPE. However, they can be made available to other SPUs or to other devices in the system.

In addition to the MFC Command Status Register, the MFC provides registers for determining the number of command slots available in the MFC proxy command queue as well as registers for determining when a tag group in the MFC proxy command queue is complete.

8.3.1 MFC Command Status Register (MFC_CMDStatus)

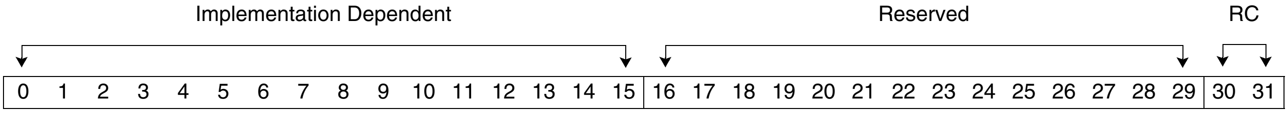
The MFC Command Status Register contains the return code from the last attempt to enqueue an MFC command to the MFC proxy command queue. The return code is read from the same location as the command was written; therefore, no intervening **eieio** instruction or **mfceieio** command is required.

Ordering of the MMIO accesses is maintained since the MFC Command Opcode Register (see page 71) and MFC Command Status Register (see page 80) are mapped to the same address.

Note: The MFC Command Status Register is a 32-bit register (the upper 16 bits are implementation-dependent). The MFC command return code in the least significant bits returns the command status when read.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘03014’; where n is the SPE number



Bits	Field Name	Description
0:15	Implementation Dependent	Implementation dependent.
16:29	Reserved	Set to zeros.
30:31	RC	MFC command return code. 00 Command enqueue successful. 01 Command enqueue failed due to sequencing error. 10 Command enqueue failed due to insufficient space in the command queue (the free space in the command queue is zero). 11 Command enqueue failed due to sequencing error, and free space in the command queue is zero.

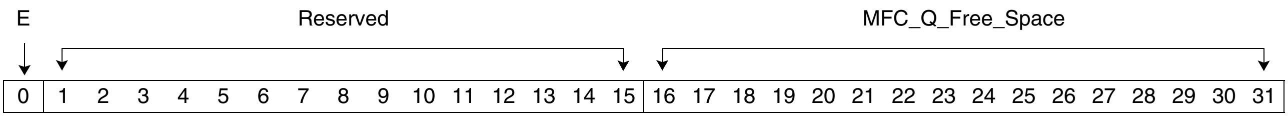
8.3.2 MFC Queue Status Register (MFC_QStatus)

The MFC Queue Status Register contains the current status of the MFC proxy command queue.
The “E” Bit (‘0’) of this register indicates either that the MFC proxy command queue is empty, or that it contains valid commands that have not finished processing. The lower 16 bits of this register return the number of entries available in the MFC proxy command queue. Zeros in these bits indicate that the queue is full.

Since the problem state registers are mapped as both caching-inhibited and guarded, an enforce-in-order-execution-of-I/O transaction (**eieio**) instruction is required between the store of a command and the load of its return code. Software must issue an eieio instruction before reading this register to ensure that the effects of all previously issued MMIO instructions or commands are reported correctly.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘03104’; where n is the SPE number



Bits	Field Name	Description
0	E	MFC proxy command queue empty. All MFC operations are complete. 0 MFC proxy command queue contains commands. 1 MFC proxy command queue does not contain any commands.
1:15	Reserved	Reserved.
16:31	MFC_Q_Free_Space	MFC proxy command queue free space. This field contains the number of queue entries available. Software can use this field to set a loop count for the number of MFC commands to enqueue. Software must not assume a command are enqueued based on the free space. Other conditions can cause the command issue sequence to fail. For proper operation, software must follow the procedure outlined in *Section 8.2 MFC Proxy Command Issue Sequence* beginning on page 78.

8.4 Proxy Tag-Group Completion Facility

Each MFC command is tagged with a 5-bit identifier, which can be used for multiple MFC commands. A set of commands with the same identifier is defined as a tag group. Software can use the identifier to determine when a command or group of commands have completed. In addition, an interrupt can be sent to a processor or device upon the completion of one or more tag groups if enabled by privileged software.

The basic procedure for polling for the completion of one or more tag groups is:

1. Issue the MFC commands to the MFC proxy command queue.
2. Set the Proxy Tag-Group Query Mask Register to the groups of interest.
3. Read the Proxy Tag-Group Status Register.
4. If the value is nonzero, one of the tag groups of interest has completed. If polling for all the tag groups of interest is complete, perform an XOR between the proxy tag-group status value and the proxy tag-group query mask; a zero indicates that all groups of interest are complete.
5. Repeat steps 3 and 4 until the tag groups of interest are complete.

The following procedure can be used to generate an interrupt when one or more tag groups complete, assuming the interrupt is enabled by privileged software. The same procedure can be used to poll for the specific condition if the Tag-Group Completion Interrupt is disabled.

1. Perform steps 1 and 2 as defined above in the polling procedure.
2. Request a tag-group query by writing a value of '01' or '11' to the Proxy Tag-Group Query Type Register.
3. Wait for an interrupt to occur if the interrupt is enabled.

OR

4. Read the Proxy Tag-Group Query Type Register until a value of zero is returned.

Notes:

1. The query type should not be changed between '01' and '10' and the query mask should not be altered until the completion condition has been met. Doing so can produce an unexpected result. The query type can be set to zero to cancel an outstanding query request.
2. Privileged software must reset the interrupt status after the interrupt is received. A new interrupt is not generated until the interrupt status is reset and the MFC re-enabled by writing the Proxy Tag-Group Query Type Register to a '01' or '10'.

8.4.1 Proxy Tag-Group Query Type Register (Prxy_QueryType)

The Proxy Tag-Group Query Type Register is used by software to request the MFC to detect a tag-group completion condition. There are two possible conditions that can be requested: the completion of *any* enabled tag groups, and the completion of *all* enabled tag groups. If enabled, a Tag-Group Completion Interrupt (see page 246), is sent to any processor, or any device in the system. See *Section 21 Interrupt Facilities* beginning on page 237 for more information.

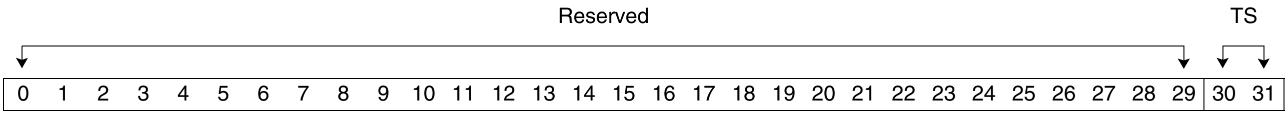
A read of this register returns the current status of the query. A nonzero value indicates that the completion condition has not occurred. When the completion condition is met, a read of this register returns zero, which indicates “query request complete.” If enabled by privileged software, a Tag-Group Completion Interrupt can be generated when the condition is met.

Writing zeros to this register while a request is still pending cancels the query request. Cancelling a query request also prevents a Tag-Group Completion Interrupt from occurring when the condition is met.

Software should not change the query type while a request is pending, except to cancel the request. Doing so can produce unexpected results.

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘03204’; where n is the SPE number



Bits	Field Name	Description
0:29	Reserved	Reserved.
30:31	TS	Tag status update condition. 00 No query requested. 01 Set interrupt status upon completion of *any* enabled tag groups. 10 Set interrupt status only upon completion of *all* enabled tag groups. 11 Reserved.

8.4.2 Proxy Tag-Group Query Mask Register (Prxy_QueryMask)

The Proxy Tag-Group Query Mask Register selects the tag groups to be included in the query operation.

The data provided by this register is retained by the MFC until changed by a subsequent write to this register. Therefore, the data does not need to be respecified for each status query. If this mask is modified by software when an MFC tag status update request is pending, the meaning of the results is ambiguous. A pending MFC tag status update request should be cancelled before a modification of this mask.

An MFC tag status update request can be cancelled by writing a value of ‘0’ (query complete) to the Proxy Tag-Group Query Type Register (see page 83).

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘0321C’; where n is the SPE number

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_F	g_E	g_D	g_C	g_B	g_A	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” select. 0 Tag group is not part of a query or status operation. 1 Tag group is part of a query or status operation.

8.4.3 Proxy Tag-Group Status Register (Prxy_TagStatus)

The Proxy Tag-Group Status Register contains the current status of the tag groups enabled in the Proxy Tag-Group Query Mask Register (see page 84). Only the status of the enabled tag groups is valid. The bit positions corresponding to disabled tag groups are set to '0'.

Software must issue an **eieio** instruction before reading this register to ensure the effects of all previous stores are reflected in the read data

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘0322C’; where n is the SPE number

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_F	g_E	g_D	g_C	g_B	g_A	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” status. 0 Tag group has outstanding operations or is not part of query (that is, it is masked). 1 Tag group is complete; no outstanding operations.

8.5 SPU Control and Status Facilities

The following SPU control and status facilities are provided:

- SPU Run Control Register
- SPU Status Register (see page 87)
- SPU Next Program Counter Register (see page 89).

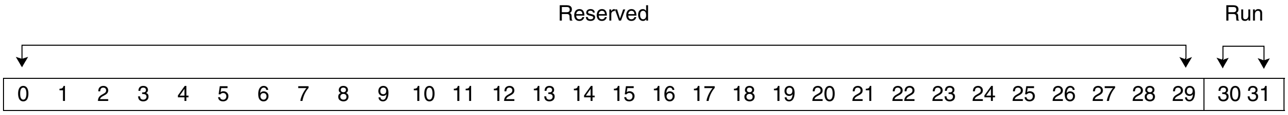
8.5.1 SPU Run Control Register (SPU_RunCntl)

The SPU Run Control Register is used to start and stop the execution of instructions in the SPU. The SPU can dynamically change the state of the Run Status bit (that is, SPU_Status[R]).

The current status of the SPU run state is available in the SPU Status Register (see page 87).

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x'0401C'; where n is the SPE number



Bits	Field Name	Description
0:29	Reserved	Set to zeros.
30:31	Run	SPU run control. 00 SPU stop request (no instructions are issued) 01 SPU run request (instructions are issued if not stalled on condition) 10 SPU isolated exit request (an exit request is ignored if the SPU is not in a stopped or halted state). 11 SPU isolates load request (a load request is ignored if the SPU is not in a stopped or halted state, or if the load request enable is not set). **Note:** The current status of the SPU run state is available in the SPU Status Register. If the Isolated state facility is not present, '10' and '11' are ignored.

8.5.2 SPU Status Register (SPU_Status)

The SPU Status Register is used to report the status (state) of an SPU. Software must issue an **eieio** instruction before reading this register to ensure that the effects of all previous stores are reflected in the read data.

Access Type	Read
Base Address Offset	(BP_Base I PS(n)) + x‘04024’; where n is the SPE number

StopCode																Reserved					E	L	Reserved	IS	C	I	S	W	H	P	R
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:15	StopCode	SPU stop and signal type code. If the P bit (stop and signal indication) is set to a ‘1’, this field provides a copy of bits 18 through 31 of the SPU stop-and-signal instruction that resulted in the SPU stop. Bits 0 and 1 of this field always are ‘0’s. If the P bit is not set, data in bits 0 through 15 is not valid. A stop-and-signal with a dependencies (STOPD) instruction, used for debugging, always sets this field to x'3FFF'.
16:20	Reserved	Reserved.
21	E	See *Section 11 SPU Isolation Facility* beginning on page 163. SPU Isolated Exit Status 0 SPU is not performing an isolate exit function. 1 SPU is performing an isolate exit function.
22	L	See *Section 11 SPU Isolation Facility* beginning on page 163. SPU Isolated Load Status 0 SPU is not performing an isolated load function. 1 SPU is performing an isolated load function.
23	Reserved	
24	IS	See *Section 11 SPU Isolation Facility* beginning on page 163. SPU Isolated State 0 SPU is not in an isolated state. 1 SPU is in an isolated state.
25	C	Invalid channel instruction detected. See note in *Section 9 Synergistic Processor Unit Channels* on page 99 for definition of an invalid channel instruction. 0 No invalid channel instruction detected. 1 Invalid channel instruction detected. SPU halted. MFC notified of error condition.
26	I	Invalid instruction detected. 0 No invalid instruction detected. 1 Invalid instruction detected. SPU stopped imprecisely. MFC notified of error condition.
27	S	SPU single-step status. See *Section 16.1 SPU Privileged Control Register* beginning on page 215. 0 SPU not stopped due to single-step mode. 1 SPU stopped after completion of an instruction in single-step mode.
28	W	SPU wait status. 0 SPU not waiting on a blocked channel. 1 SPU waiting on a blocked channel.
29	H	SPU halt instruction status. 0 SPU not halted (imprecise stopped) due to a halt instruction. 1 SPU halted (imprecise stopped) due to a halt instruction.

Bits	Field Name	Description
30	P	SPU program stop-and-signal status. (Applies to either STOP or STOPD instruction.) 0 SPU not stopped due to stop-and-signal instruction. 1 SPU stopped due to stop-and-signal instruction.
31	R	SPU run status. 0 SPU stopped or halted. 1 SPU running.

Programming Note:

A read of this register provides a current view of the SPU state. The status read can change dynamically, if the SPU is running. When the SPU is restarted bits, C, I, S, H, and P bits are cleared (that means that software does not have to clear these bits before restarting).

If the SPU is stopped, the status remains static until it is changed by software. If the SPU was stopped under PPE control while waiting on a blocked channel, the SPU wait status is set in conjunction with the SPU stopped status.

When a stop-and-signal instruction is executed by the SPU, the least-significant 14 bits of the instruction are copied to bits 2 through 15 of the SPU Status Register as a STOP type code. Bits ‘0’ and ‘1’ of the SPU Status Register are always zeros. It is recommended that a value of x‘0’ for the STOP type be reserved for “DATA Executed as Instruction. It is further recommended that type values having the most significant bit set (bit 2) be reserved for runtime or privileged services.

A type value of x‘3FFF’ should be reserved for debugger breakpoints, since that value is hard wired on a **STOPD** instruction.

Type codes with bit 2 of the type code set to ‘0’s (x‘0001’ - x‘1FFF’) are available for application use. These definitions are not enforced, but are recommended conventions that could eventually become part of the CBEA-compliant application binary interface (ABI).

8.5.3 SPU Next Program Counter Register (SPU_NPC)

The SPU Next Program Counter Register contains the address from which an SPU starts executing in response to a set of the run control bit in the SPU Run Control Register (see page 86).

Access to this register is available in all states. Writes to this register update the contents only when the SPU is in the stopped and nonisolated state (SPU_Status[R] = '0', SPU_Status[IS] = '0']). Writes to this register have no effect if the SPU is an isolated state (SPU_Status[IS] = '1') or if the SPU is currently in the running state. When the SPU is in the nonisolated and stopped state, a read of this register returns the local storage address of the next instruction to be executed if the SPU is started without an intervening write to this register. In addition, the least significant bit of the data returned indicates the starting SPU interrupt enable or disable state. When a read of this register is performed while the SPU is in an isolated state, a value of all zeros is always returned. The data returned for read of this register when the SPU is in the running and nonisolated state (SPU_Status[R] = '1', SPU_Status[IS] = '0']) is undefined.

When the SPU has stopped execution, the hardware automatically updates the value in this register with the address of the next instruction to be executed and with the current SPU interrupt enabled or disabled state if execution is to be resumed with an SPU start command. The SPU can be stopped either by the execution of SPU conditional halt instructions, or by an SPU error, or by the execution of an SPU stop-and-signal instruction, or by the execution of a single instruction step, or by resetting the RUN control bit in the SPU Run-Control Register. If the stop was due to a stop-and-signal instruction in nonisolated state, the location of the actual stop-and-signal instruction can be determined by masking the enable or disable interrupt state bit (LSb) then subtracting four from the value read from this register. For proper operation, software must ensure that the SPU program execution has stopped in a nonisolated state by reading and checking the values in the SPU Status Register before reading the SPU Next Program Counter Register.

To resume execution of a program, software writes the SPU Run Control Register to the appropriate value, or issues an MFC command with the S (start SPU) option. In a nonisolated state, to resume execution at a different point in the program or if a new program is loaded into the SPU, this register should be written by software to set the SPU program counter to the address in local storage of the next to be executed, and the initial interrupt enable or disable state to be effective when the SPU is started. It is not possible to affect the SPU execution address or state with this facility when the SPU is in an isolated state.

Access Type Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘04034’; where n is the SPE number

Local Storage Address																														NI	IE
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:29	Local Storage Address	This field contains the local storage address of the first instruction issued when the run bit is enabled.
30	NI	Reserved
31	IE	Interrupt enable state. 0 SPU interrupts disabled at start. 1 SPU interrupts enabled at start.

8.6 Mailbox Facility

The MFC provides a set of mailbox queues between the SPU and other processors and devices. Each mailbox queue has an SPU channel assigned as well as a corresponding MMIO register. SPU software accesses the mailbox queues by using SPU channel instructions; other processors and devices access the mailbox queues by using one of the MMIO registers. In addition to the queues, the MFC provides queue status, mailbox interrupts, and SPU event notification for the mailboxes. Collectively, the MMIO registers, channels, status, interrupts, mailbox queues, and events are referred to as the Mailbox Facility.

Two mailbox queues are provided by the MFC for sending information from the SPU to another processor or to other devices: the SPU Outbound Mailbox queue and the SPU Outbound Interrupt Mailbox queue. These mailbox queues are intended for sending short messages to the PPE (for example, return codes or status).

Data written by the SPU to one of these queues using a write channel (**wrch**) instruction is available to any processor, or device by reading the corresponding MMIO register. A write channel (**wrch**) instruction sent to the SPU Write Outbound Interrupt Mailbox Channel (see page 123), also can cause an interrupt to be sent to a processor, or to another device in the system.

An MMIO read from either of these queues (SPU Outbound Mailbox or SPU Outbound Interrupt Mailbox registers) can set an SPU event, which in turn causes an SPU interrupt. See *Section 9.11 SPU Event Facility* beginning on page 133 for a description of the SPU Event Facility and *Section 9.12 SPU Event Definitions* beginning on page 145 for a description of the events that can be caused.

One mailbox queue is provided for either an external processor or other devices to send information to the SPU: the SPU Inbound Mailbox queue. This mailbox queue is intended to be written by the PPE. However, other processors, SPUs, or other devices can also use this mailbox queue. Data written by a processor or another device to this queue using an MMIO write is available to the SPU by reading the SPU Read Inbound Mailbox Channel (see page 124).

An MMIO write to the SPU Inbound Mailbox Register can set an SPU event, which in turn can cause an SPU interrupt.

The rest of this section describes the following registers:

- SPU Outbound Mailbox Register (see page 91)
- SPU Inbound Mailbox Register (see page 92)
- SPU Mailbox Status Register (see page 93)

For information on the privilege 2 mailbox interrupt MMIO register and the channels that correspond to these registers, see:

- *Section 15.12 SPU Outbound Interrupt Mailbox Register* beginning on page 213
- *Section 9.5.1 SPU Write Outbound Mailbox Channel* beginning on page 122
- *Section 9.5.2 SPU Write Outbound Interrupt Mailbox Channel* beginning on page 123
- *Section 9.5.3 SPU Read Inbound Mailbox Channel* beginning on page 124

8.6.1 SPU Outbound Mailbox Register (SPU_Out_Mbox)

The SPU Outbound Mailbox Register is used to read 32 bits of data from the corresponding SPU outbound mailbox queue. The SPU Outbound Mailbox Register has a corresponding SPU Write Outbound Mailbox Channel (see page 122), for writing data into the SPU outbound mailbox queue.

A write channel (**wrch**) instruction sent to the SPU outbound mailbox queue loads the 32 bits of data specified in the instruction into the SPU outbound mailbox queue for other processors or other devices to read. If the SPU outbound mailbox queue is full, the SPU stalls on the write channel (**wrch**) instruction that is sent to this queue until an MMIO read from this mailbox register occurs.

An MMIO read of this register always returns the information in the order that it was written by the SPU. The information returned on a read from an empty SPU outbound mailbox queue is undefined.

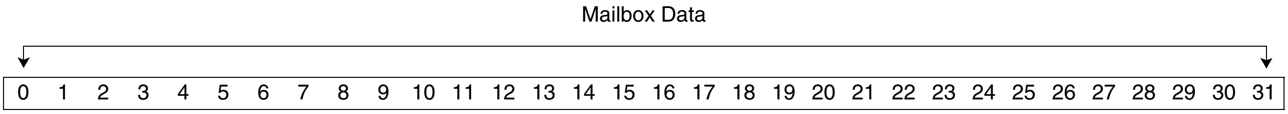
The number of entries in the SPU outbound mailbox queue (or queue depth) is implementation-dependent.

An MMIO read of the SPU Mailbox Status Register (see page 93) returns the status of the mailbox queues. The number of valid queue entries in the SPU outbound mailbox queue is given in the SPU_Out_Mbox_Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat [SPU_Out_Mbox_Count]).

An MMIO read of the SPU Outbound Mailbox Register sets a pending SPU Outbound Mailbox Available Event. If the amount of data remaining in the mailbox queue is below an implementation-dependent threshold and this condition is enabled (that is, SPU_WrEventMask[Le] set to ‘1’), then the SPU Read Event Status Channel is updated (that is, SPU_RdEventStat[Le] is set to ‘1’), and its channel count is set to ‘1’, which causes an SPU Outbound Interrupt Mailbox Available Event (see page 150).

Access Type MMIO: Read

Base Address Offset (BP_Base | PS(n)) + x‘04004’; where n is the SPE number



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge a write to the SPU Write Outbound Mailbox Channel (see page 122) until the PPE or other device has read the contents of the mailbox.

8.6.2 SPU Inbound Mailbox Register (SPU_In_Mbox)

The SPU Inbound Mailbox Register is used to write 32 bits of data into the corresponding SPU inbound mailbox queue. The SPU inbound mailbox queue has the corresponding SPU Read Inbound Mailbox Channel (see page 124) for reading data from the queue.

A read channel (**rdch**) instruction of the SPU Read Inbound Mailbox Channel loads the 32 bits of data from the SPU inbound mailbox queue into the SPU register specified by the read channel (**rdch**) instruction. The SPU cannot read from an empty mailbox. If the SPU inbound mailbox queue is empty, the SPU stalls on a read channel (**rdch**) instruction to this channel until data is written to the mailbox. A read channel (**rdch**) instruction to this channel always returns the information in the order that it was written by the PPE or by other processors and devices.

The number of entries in the queue (or queue depth) is implementation-dependent.

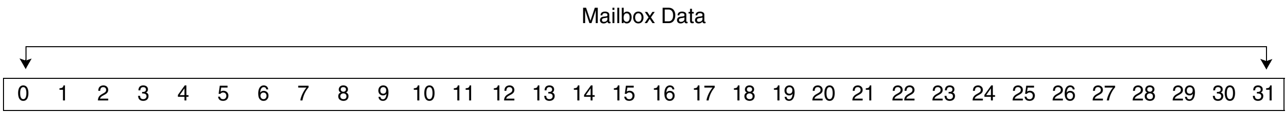
An MMIO read of the SPU Mailbox Status Register (see page 93), returns the state of the mailbox queues. The number of available queue locations in the SPU mailbox queue is given in the SPU_In_Mbox_Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat[SPU_In_Mbox_Count]).

Software should check the SPU Mailbox Status Register before writing to the SPU_In_Mbox to avoid overrunning the SPU mailbox.

An MMIO write of the SPU Inbound Mailbox Register sets a pending SPU mailbox event. If enabled (that is, SPU_WrEventMask[Mbox] =’1’), the SPU Read Event Status Channel is updated and its channel count is set to ‘1’, which causes an SPU Inbound Mailbox Available Event (see page 148).

Access Type MMIO: Write Only[1]

Base Address Offset (BP_Base | PS(n)) + x‘0400C’; where n is the SPE number



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

[1] Read is only supported for diagnostic purposes.

8.6.3 SPU Mailbox Status Register (SPU_Mbox_Stat)

The SPU Mailbox Status Register contains the current state of the mailbox queues between the SPU and the PPE in the corresponding SPE. Reading this register has no effect on the state of the mailbox queues.

Access Type Read

Base Address Offset (BP_Base | PS(n)) + x‘04014’; where n is the SPE number

Reserved								SPU_Out_Intr_Mbox_ Count								SPU_In_Mbox_Count								SPU_Out_Mbox_Count							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:7	Reserved	Set to zeros.
8:15	SPU_Out_Intr_Mbox_ Count	Number of valid entries in the SPU outbound interrupt mailbox queue (optional for a one-deep interrupt mailbox) The SPU_Out_Intr_Mbox_ Count value is incremented when the SPU writes the SPU Write Outbound Interrupt Mailbox Channel (see page 123) and is decremented when a processor or other device reads the SPU Outbound Interrupt Mailbox Register (see page 213). The number of entries supported is implementation-dependent.
16:23	SPU_In_Mbox_Count	Number of available entries in the SPU Inbound Mailbox queue The SPU_In_Mbox_Count value is decremented when a processor or other device writes the SPU Inbound Mailbox Register, and is incremented when the SPU reads the SPU Read Inbound Mailbox Channel. The number of entries supported is implementation-dependent.
24:31	SPU_Out_Mbox_Count	Number of valid entries in the SPU outbound mailbox queue The SPU_Out_Mbox_Count value is incremented when the SPU writes the SPU Write Outbound Mailbox Channel and is decremented when a processor or other device reads the SPU Outbound Mailbox Register. The number of entries supported is implementation-dependent.

8.7 SPU Signal Notification Facility

The SPU Signal Notification Facility is used to send signals, such as a buffer completion flag to an SPU from other processors and devices in the system. The CBEA provides two independent signal notification facilities: SPU Signal Notification 1 and SPU Signal Notification 2.

Each facility consist of one register and one channel:

- SPU Signal Notification 1 Register (see below) and SPU Signal Notification 1 Channel (see page 126)
- SPU Signal Notification 2 Register (see page 95) and SPU Signal Notification 2 Channel (see page 127).

Signals are issued by an SPU using a set of send signal commands (**sndsig*[<f,b>]***) with the effective address of the Signal Notification Register associated with the SPU to which the signal is sent.
(For information on the commands, see *Section 7.9.4 Send Signal Command* on page 66 and *Section 7.9.5 Send Signal with Fence or with Barrier Command* on page 67.)

PPEs and other devices that do not support send signal commands, simulate sending a signal command by performing an MMIO write to the SPU Signal Notification Register associated with the SPU to which the signal is to be sent. These registers are described in this section.

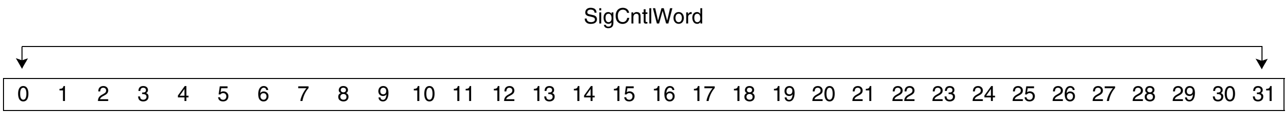
Each of the signal notification facilities can be programmed to either an overwrite mode, which is useful in a one-to-one signalling environment or to a logical OR mode, which is useful in a many-to-one signalling environment. The mode for each channel is set in the SPU Configuration Register (see page 221). Performing either a send signal command or an MMIO that targets a signalling register programmed to overwrite mode sets the contents of the associated channel to the data of the signalling operation. It also sets the corresponding channel count to '1'. In logical OR mode, the data of the signalling operation is ORed with the current contents of the channel, and the corresponding count is set to a value of '1'.

In addition, the signal notification registers are used as the effective address of an image, when performing an isolated load. For more information, see *Section 11 SPU Isolation Facility* beginning on page 163. In these cases, SPU Signal Notification 1 Register contains the upper 32 bits of the 64-bit effective address, and SPU Signal Notification 2 Register contains the least significant 32 bits. Software must have the SPU Signal Notification Facility in an overwrite mode before setting the effective address for proper operation of an isolated load request.

8.7.1 SPU Signal Notification 1 Register (SPU_Sig_Notify_1)

Access Type MMIO: Read/Write

Base Address Offset (BP_Base | PS(n)) + x'1400C'; where n is the SPE number

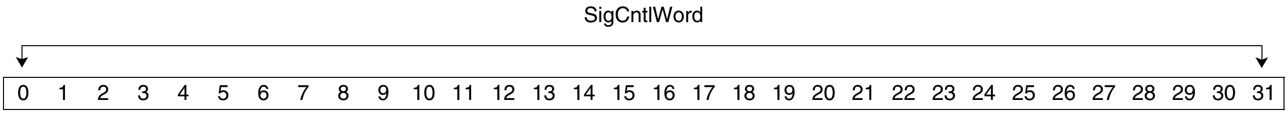


Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word The data is defined by the application. This can either be ORed with the previous value, or it can overwrite the previous value. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 221.

8.7.2 SPU Signal Notification 2 Register (SPU_Sig_Notify_2)

Access Type MMIO: Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘1C00C’; where n is the SPE number



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word The data is defined by the application. It can either be ORed with the previous value or it can be overwritten. For more information, see *Section 16.4 SPU Configuration Register* beginning on page 221.

8.8 MFC Multisource Synchronization Facility

The MFC Multisource Synchronization Facility achieves cumulative ordering across the local storage and main storage address domains. Standard PowerPC ordering rules apply to storage accesses performed by one processor or unit with respect to another processor or unit. Ordering of storage accesses performed by multiple sources (that is, two or more processors or units) with respect to another processor or unit is referred to as cumulative ordering. See *PowerPC Architecture, Book II* for additional details.

Cumulative ordering is ensured when all accesses are performed in the main storage address domain and the proper synchronization instructions are performed. Cumulative ordering cannot be ensured when accesses are performed from both the main storage and the local storage address domains.

The MFC Multisource Synchronization Facility consists of:

- The MFC Multisource Synchronization Register, which allows processors or devices to control synchronization from the main storage address domain
- The MFC Write Multisource Synchronization Request Channel (see page 132), which allows an SPU to control synchronization from the local storage address domain.

Each ensures that all transfers are sent where the destination is the associated MFC (that is, write transfers sent to the MFC) and received before the MFC multisource synchronization requests are completed. This facility does not ensure that read data is visible at the destination when the associated MFC is the source.

8.8.1 MFC Multisource Synchronization Register (MFC_MSSync)

The MFC Multisource Synchronization Register is part of the MFC Multisource Synchronization Facility. Writing any value to this register requests a synchronization. At the time of the write, the MFC starts to track all outstanding transfers sent to the corresponding SPE. When read, this register returns the current status of the last request.

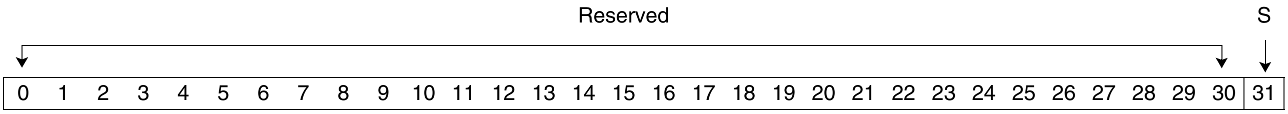
A value of zero is returned when all transfers sent to the SPE and received before the last write of the MFC Multisource Synchronization Register is complete.

To use the MFC Multisource Synchronization Register, a program must:

1. Write to the MFC Multisource Synchronization Register.
2. Poll the MFC Multisource Synchronization Register until a value of 0 is received.

Access Type MMIO: Read/Write

Base Address Offset (BP_Base | PS(n)) + x‘00000’; where n is the SPE number



Bits	Field Name	Description
0:30	Reserved	Set to zeros.
31	S	MFC Multisource Synchronization Status 0 All transfers received before MFC_MSSync register write are complete. 1 All transfers received before MFC_MSSync register write are not complete.

Programming Note:

Use of the MFC Multisource Synchronization Register is required for swapping context on an SPE. After stopping the SPU, privileged software must prevent any new transfers from being initiated to the SPE by unmapping the associated resources. Next, privileged software must use the MMIO MFC Multisource Synchronization Register to ensure the completion of all outstanding transfers. In addition, this has the side effect of ensuring the MFC Write Multisource Synchronization Request Channel count is '1' and the SPU Event Status is updated.

The following are examples of when the different Multisource Synchronization facilities are required. These examples show an I/O device storing data to the local storage alias area followed by a store to main storage. A second processor reads main storage and stores to another location in the local storage alias area. Cumulative ordering requires the value loaded from location X by the SPU must be 1.

The first example illustrates the use of the MFC Multisource Synchronization Facility and the second is an illustration of the MFC Write Multisource Synchronization Request Channel. In both of these examples, the locations X, Y, and Z all have an initial value of zero.

A third example is provided to illustrate the use of the MFC Multisource Synchronization Facility when starting an SPU.

Without the use of these facilities, the SPU might not receive a '1' when reading the local storage location X.

Example 1 (MFC Multisource Synchronization Facility**):**

I/O Device (I/O transfers are always processed in order)
Stores '1' to local storage alias X
Stores '2' to main storage location Y.
Processor
Loops loading from main storage location Y until a value of 2 is obtained
Stores to MFC Multisource Synchronization Register
Polls MFC Multisource Synchronization Register until a zero is obtained
Stores '3' to local storage alias Z.
SPU
Loops loading from local storage location Z until a value of '3' is obtained then loads from local storage location X and obtains '1'.

Example 2 (MFC Write Multisource Synchronization Request Channel):

I/O Device (I/O transfers are always processed in order)
Stores '1' to local storage alias X
Stores '2' to main storage location Y
Processor
Loops loading from main storage location Y until a value of '2' is obtained then stores '3' to local storage alias 'Z'
SPU
Loads from local storage location 'Z' and obtains '3'
Writes to channel 2 with bit 19 set to acknowledge a previous Multisource Synchronization Event
Writes to channel 1 with bit 19 set to enable the Multisource Synchronization Event
Channel writes to MFC Write Multisource Synchronization Request Channell
Wait on Multisource Synchronization Event to occur

{Wait can be coded as a read of the count for MFC_MSSyncReq channel, reading the SPU Read Event Status Channel (see page 136), or with a **bisled** instruction}

Loads from local storage location X and obtains 1.

Example 3 (MFC Multisource Synchronization Facility):

I/O Device (I/O transfers are always processed in order)

Stores ‘1’ to local storage alias X

Interrupts Processor A

Processor A

After receiving the interrupt, stores to MFC Multisource Synchronization Register

Polls MFC Multisource Synchronization Register until a zero is obtained

Stores x'01' to the SPU Run Control Register to start the SPU.

SPU

SPU starts to execute instructions, which load from local storage location X and obtains ‘1’.

Note: Frequent use of a single MFC Multisource Synchronization Facility by two or more processors or devices can result in a livelock condition. The livelock occurs when a read from a processor or from a device never returns zero due to a synchronization request from other processors or devices.

Implementation Note:

The MFC Multisource Synchronization Facility and MFC Write Multisource Synchronization Request Channel must be treated independently. A synchronization request from the MFC proxy facility must not have any effects on the channel request and vice versa.

9. Synergistic Processor Unit Channels

In the Cell Broadband Engine Architecture (CBEA), channels are used as the primary interface between the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPU Channel Access Facility (see page 218) is used to configure, save, and restore the SPU channels. The SPU Instruction Set Architecture (ISA) provides a set of channel instructions for communication with external devices through a channel interface (or SPU channels). *Table 9-1* lists these instructions.

Table 9-1. SPU Channel Instructions

Channel Instruction	Instruction Mnemonic	Operational Description
Read Channel	**rdch**	Causes a read of data stored in the addressed channel to be loaded into the selected General Purpose Register (GPR).
Write Channel	**wrch**	Causes data to be read from the selected GPR and stored in the addressed channel.
Read Channel Count	**rchcnt**	Causes the count associated with the addressed channel to be stored in the selected GPR.

Architecturally, SPU channels can be configured to be read-only or write-only; channels cannot be configured as read and write. However, all channels have an associated channel count that can always be read by issuing the read channel count (**rchcnt**) instruction. In addition to the access type, each channel can be configured as nonblocking or blocking. Channels that are configured as blocking cause the SPU to stall when reading a channel with a channel count of ‘0’, or writing to a full channel (that is, a channel with a channel count of ‘0’).

- Read—Means that only a read channel instruction (**rdch**) can be issued to this channel and data is always returned.
- Write—Means that only a write channel instruction (**wrch**) can be issued to this channel and data is always accepted by the channel.
- Read-blocking—Means that only a read channel instruction (**rdch**) can be issued to this channel.
  A read channel instruction (**rdch**) sent to a read-blocking channel only completes if the channel count is not zero. A channel count of ‘0’ indicates that the channel is empty. Executing a channel read (**rdch**) to a read-blocking channel with a count of ‘0’ results in the SPU stalling until data is available in the channel.
- Write-blocking—Means that only a write channel instruction (**wrch**) can be issued to this channel.
  A write channel (**wrch**) instruction sent to a write-blocking channel only completes if the channel count is not zero. A channel count of ‘0’ indicates that the channel is full. Executing a write channel (**wrch**) instruction to a write-blocking channel with a count of ‘0’ results in the SPU stalling until an entry in the addressed channel becomes available.

Note: Issuing a channel instruction that is inappropriate to the configuration of the channel results in an invalid channel instruction interrupt. For example, issuing a read channel instruction (**rdch**) to a channel configured as a write or a write-blocking channel results in an invalid channel instruction interrupt.

Each channel has a corresponding count (that is, depth), which indicates the number of outstanding operations that can be issued for that channel. The channel depth (that is, the maximum number of outstanding transfers) is implementation-dependent. Software must initialize the channel counts when establishing a new context in the SPU, or when it resumes an existing context.

Table 9-2 on page 100 shows the assignments and configurations of SPU channels to various facilities within the MFC. Each SPU in the processor contains the full set of channels outlined in *Table 9-2* and an SPU Channel Access Facility (see page 218).

Note: No reserved channels can be used for implementation-dependent functions.

Table 9-2. SPU Channel Map (Page 1 of 2)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
SPU Event Channels			
x‘0’	SPU_RdEventStat	SPU Read Event Status Channel (see page 136) Read event status (with mask applied).	Read-blocking
x‘1’	SPU_WrEventMask	SPU Write Event Mask Channel (see page 140) Write event-status mask.	Write
x‘2’	SPU_WrEventAck	SPU Write Event Acknowledgment Channel (see page 144) Write end-of-event processing.	Write
SPU Signal Notification Channels			
x‘3’	SPU_RdSigNotify1	SPU Signal Notification 1 Channel (see page 126)	Read-blocking
x‘4’	SPU_RdSigNotify2	SPU Signal Notification 2 Channel (see page 127)	Read-blocking
x‘5’	Channel 5	Reserved	
x‘6’	Channel 6	Reserved	
SPU Decrementer Channels			
x‘7’	SPU_WrDec	SPU Write Decrementer Channel (see page 128)	Write
x‘8’	SPU_RdDec	SPU Read Decrementer Channel (see page 129)	Read
MFC Multisource Synchronization Channels			
x‘9’	MFC_WrMSSyncReq	MFC Write Multisource Synchronization Request Channel (see page 132)	Write-blocking
SPU Reserved Channel			
x‘A’	Channel 10	Reserved	
Mask Read Channels			
x‘B’	SPU_RdEventMask	SPU Read Event Mask Channel (see page 142)	Read
x‘C’	MFC_RdTagMask	MFC Read Tag-Group Query Mask Channel (see page 115)	Read
SPU State Management Channels			
x‘D’	SPU_RdMachStat	SPU Read Machine Status Channel (see page 130)	Read
x‘E’	SPU_WrSRR0	SPU Write State Save-and-Restore Channel (see page 131)	Write
x‘F’	SPU_RdSRR0	SPU Read State Save-and-Restore Channel (see page 131)	Read
MFC Command Parameter Channels			
x‘10’	MFC_LSA	MFC Local Storage Address Channel (see page 107) Write local storage address command parameter.	Write
x‘11’	MFC_EAH	MFC Effective Address High Channel (see page 109) Write high-order MFC effective-address command parameter.	Write
x‘12’	MFC_EAL	MFC Effective Address Low or List Address Channel (see page 108) Write low-order MFC effective-address command parameter.	Write

Table 9-2. SPU Channel Map (Page 2 of 2)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
x'13'	MFC_Size	MFC Transfer Size or List Size Channel (see page 106) Write MFC transfer size command parameter.	Write
x'14'	MFC_TagID	MFC Command Tag Identification Channel (see page 105) Write TAG identifier command parameter.	Write
x'15'	MFC_Cmd MFC_ClassID	MFC Command Opcode Channel (see page 103) Write and enqueue MFC command with associated class ID.	Write-blocking
		MFC Class ID Channel (see page 103) Write and enqueue MFC command with associated command opcode.	
MFC Tag Status Channels			
x'16'	MFC_WrTagMask	MFC Write Tag-Group Query Mask Channel (see page 114) Write TAG mask.	Write
x'17'	MFC_WrTagUpdate	MFC Write Tag Status Update Request Channel (see page 116) Write request for conditional or unconditional tag status update.	Write-blocking
x'18'	MFC_RdTagStat	MFC Read Tag-Group Status Channel (see page 117) Read TAG status (with mask applied).	Read-blocking
x'19'	MFC_RdListStallStat	MFC Read List Stall-and-Notify Tag Status Channel (see page 118) Read MFC list stall-and-notify status.	Read-blocking
x'1A'	MFC_WrListStallAck	MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119) Write MFC list stall-and-notify acknowledgment.	Write
x'1B'	MFC_RdAtomicStat	MFC Read Atomic Command Status Channel (see page 120) Read atomic command status.	Read-blocking
SPU Mailboxes			
x'1C'	SPU_WrOutMbox	SPU Write Outbound Mailbox Channel (see page 122) Write outbound SPU mailbox contents.	Write-blocking
x'1D'	SPU_RdInMbox	SPU Read Inbound Mailbox Channel (see page 124) Read inbound SPU mailbox contents.	Read-blocking
x'1E'	SPU_WrOutIntrMbox	SPU Write Outbound Interrupt Mailbox Channel (see page 123) Write SPU outbound interrupt mailbox contents.	Write-blocking
x'1F' — x'3F'	Channel 31 — Channel 63	Reserved	

9.1 MFC SPU Command Parameter Channels

The MFC command parameter registers are described in *Section 8.1 MFC Command Parameter Registers* beginning on page 70. The channels used to queue an MFC command to the MFC SPU command queue are described in the following sections.

Specifically:

- *Section 9.1.1 MFC Command Opcode Channel* beginning on page 103 describes using channel x‘15’ (lower bits).
- *Section 9.1.2 MFC Class ID Channel* beginning on page 103 describes using channel x‘15’ (upper bits).
- *Section 9.1.3 MFC Command Tag Identification Channel* beginning on page 105 describes using channel x‘14’.
- *Section 9.1.4 MFC Transfer Size or List Size Channel* beginning on page 106 describes using channel x‘13’.
- *Section 9.1.5 MFC Local Storage Address Channel* beginning on page 107 describes using channel x‘10’.
- *Section 9.1.6 MFC Effective Address Low or List Address Channel* beginning on page 108 describes using channel x‘12’.
- *Section 9.1.7 MFC Effective Address High Channel* beginning on page 109 describes using channel x‘11’.

The MFC command parameter channels are nonblocking and do not have channel counts associated with them. Performing a read channel count (**rchcnt**) instruction that is sent to any of these channels returns a count of ‘1’.

The MFC Command Opcode Channel has a maximum count configured by hardware to the number of MFC queue commands supported by the hardware. Software must initialize the MFC Command Opcode Channel count to the number of empty MFC command queue slots supported by the implementation after power-on, and after a purge of the MFC queue. This channel count must also be saved and restored on an SPU Element (SPE) preemptive context switch.

Implementation Note:

An MFC command must not be put into the command queue until a write channel (**wrch**) of the MFC command to MFC Command Opcode Channel is executed by the SPU. The maximum count of the MFC Command Opcode Channel should be configured by hardware to the number of empty MFC command slots supported by the implementation.

9.1.1 MFC Command Opcode Channel (MFC_Cmd)

The MFC Command Opcode Channel determines the operation to be performed. The validity of this opcode is checked asynchronously to the instruction stream. If the MFC command, or any of the command parameters are invalid, MFC command queue processing is suspended and an invalid MFC command interrupt is sent. For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

The MFC command and the class ID parameters must be written to the MFC SPU command queue using a single channel instruction. The MFC command parameter is the lower 16-bits of the 32-bit word. The upper 8 bits of this field are reserved.

Software must avoid programming practices that place commands in the queue with forward dependencies on newer commands placed in the queue. Software with this type of dependency can create a deadlock based on the number of available slots in the MFC queue. In addition, while queue depth is implementation dependent, software must *not* be written to require a specific queue depth.

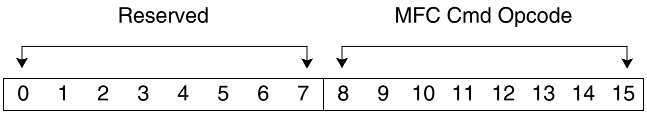
Software can determine the number of queue entries available in the MFC SPU command queue by reading the channel count (**rchcnt**) instruction of this channel. The value returned is the number of available queue slots.

Software can use this value to avoid stalling the execution of an SPU program, which occurs when an MFC command enqueue is attempted to a full queue.

The queuing sequence for MFC SPU commands is described in *Section 9.2 MFC SPU Command Issue Sequence* beginning on page 110.

Access Type Write-blocking

Channel Number x‘15’ (lower 16 bits)



Bits	Field Name	Description
0:7	Reserved	Should normally be set to zeros. Bit 0 set to ‘1’ indicates that the opcode is reserved.
8:15	MFC Cmd Opcode	MFC command opcode. See *Section 7 MFC Commands* beginning on page 47 for the MFC command opcodes.

Programming Note:

The total number of queue slots is implementation dependent and varies between implementations. For portability of an application, the enqueue sequence for DMA commands and the method to determine the number of queue slots available should be provided as a macro or library function that utilizes a configuration dependent method for queueing commands.

9.1.2 MFC Class ID Channel (MFC_ClassID)

The MFC Class ID Channel is used to specify the replacement class ID and the transfer class ID for each MFC command. These IDs are used by the processor and software to improve the overall performance of the system. The exact function and mapping of the class IDs is implementation-dependent. For more information, see *Section 19 Cache Replacement Management Facility* on page 231.

The transfer class ID (TclassID) is used to identify access to storage with differing characteristics. The TclassID is intended to be used to allow an implementation to optimize the transfers corresponding to the MFC command based on the characteristics of the storage location. Setup and use of the TclassID is implementation dependent. Refer to the specific implementation documentation.

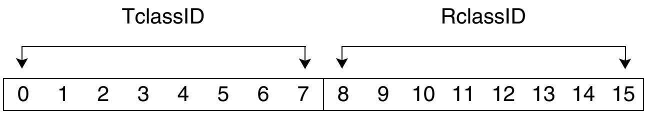
The contents of the class ID parameter are not persistent and must be written for each MFC command enqueue sequence.

The class ID parameter performs the same function, whether it is used with commands issued from the PPE or the SPU side of the command queue. This parameter is used to control resources associated with an SPE and it has no effect on resources associated with other SPEs or PPEs.

The validity of the class ID parameter is not verified. The number of class IDs supported in implementation dependent. The default class ID (x‘00’) is used for all undefined or invalid class IDs. An invalid class ID does not generate an exception.

Access Type Write-blocking

Channel Number x‘15’ (upper 16 bits)



Bits	Field Name	Description
0:7	TclassID	Transfer class identifier
8:15	RclassID	Replacement class identifier

Note: The MFC class ID parameter must be written to the MFC SPU queue along with the command parameter using a single channel. The MFC command parameter is the lower 16-bits of the 32-bit word.The MFC class ID parameter is the upper 16-bits of the 32-bit word.

9.1.3 MFC Command Tag Identification Channel (MFC_TagID)

The MFC Command Tag Identification Channel is used to specify an identifier for each command, or for a group of commands. The identification tag is any value between x‘0’ and x‘1F’. Identification tags have a purely local scope in the hardware.

Any number of MFC commands can be tagged with the same identification. MFC commands tagged with the same identification are referred to as a tag group.

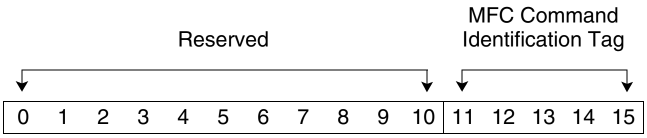
Tags are associated with commands written to a specific queue. Tags supplied to the MFC SPU command queue are independent of the tags supplied to the MFC proxy command queue.

The contents of the MFC command tag identification parameter are not persistent and must be written for each MFC command enqueue sequence.

The validity of this parameter is checked asynchronous to the instruction stream. If the upper bits (bits 0 through 10) are not set to zeros, the MFC command queue processing is suspended and an interrupt is generated.
For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Access Type Write

Channel Number x‘14’



Bits	Field Name	Description
0:10	Reserved	Set to zeros.
11:15	MFC Command Tag Identification	MFC command tag identification

9.1.4 MFC Transfer Size or List Size Channel (MFC_Size)

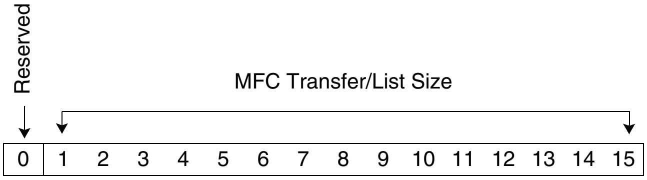
The MFC Transfer Size or List Size Channel is used to specify the size of the MFC transfer, or the size of the MFC DMA transfer list. The transfer size can have a value of 0, 1, 2, 4, 8, 16, or a multiple of 16 bytes to a maximum of 16 KB. The MFC transfer list size can have a value of 8, or of multiples of 8 up to a maximum of 16 KB.

The contents of the MFC Transfer Size or List Size Channel are not persistent and must be written for each MFC command enqueue sequence.

The validity of this parameter is checked asynchronous to the instruction stream. If the size is invalid, the MFC command queue processing is suspended and an MFC DMA alignment interrupt is sent. For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Access Type Write

Channel Number x‘13’



Bits	Field Name	Description
0	Reserved	Set to zero.
1:15	MFC Transfer/List Size	MFC transfer size or list size. Valid range is: • 0 ≤ MFC Transfer Size ≤ 16 KB • 0 ≤ MFC List Size ≤ 16 KB Allowable MFC transfer/list sizes are: • 0,1, 2, 4, and 8 bytes (naturally aligned), where the source and destination address *must* have the same four least-significant bits (see *Book II* of the *PowerPC Architecture* for more information on DMA alignment). • 16 bytes and multiples of 16 bytes up to 16 KB, where the source and destination addresses must be 16-byte (quadword) aligned. Allowable DMA list sizes are: • Multiples of 8 bytes with a minimum size of 0 bytes and a maximum size of 16 KB for list size, where the list must start on an 8-byte (doubleword) boundary in local storage. This provides a list with from 0 to 2048 possible elements.

Programming Note:

Transfers of less than one cache line should be used sparingly. Excessive use of small transfers results in poor bus and memory bandwidth utilization. Due to their low performance and high setup cost, transfers of less than 16 bytes should only be used when necessary to interface with an I/O device. When transferring 128 bytes or more, programmers should attempt to align the source and destination on 128-byte boundaries for optimal performance.

9.1.5 MFC Local Storage Address Channel (MFC_LSA)

The MFC Local Storage Address Channel is used to supply the SPU local storage address associated with an MFC command to be queued. This address is used as the source or destination of the MFC transfer as defined in the MFC command. For more information, see *Section 7 MFC Commands* beginning on page 47.

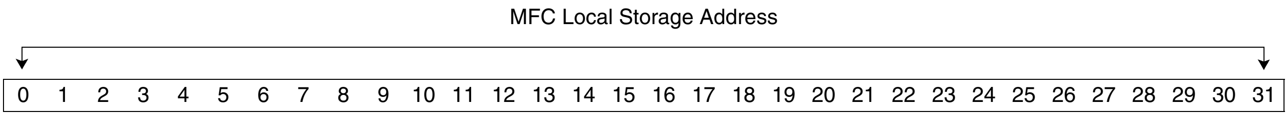
The contents of the MFC Local Storage Address Channel are not persistent and must be written for each MFC command enqueue sequence.

The validity of this parameter is checked asynchronously to the instruction stream. If the address is unaligned, MFC command queue processing is suspended, and an MFC DMA alignment exception is generated.
For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

To be considered aligned, the 4 least significant bits of the local storage address must match the least-significant 4 bits of the effective address.

Access Type Write

Channel Number x‘10’



Bits	Field Name	Description
0:31	MFC Local Storage Address	MFC local storage address **Note:** The 4 least-significant bits of the local storage address must match the least-significant 4 bits of the effective address.

Programming Note:

For optimum performance, transfers of greater than 16 bytes should have bits[25:31] of the local storage address set to 0.

9.1.6 MFC Effective Address Low or List Address Channel (MFC_EAL)

The MFC Effective Address Low or List Address Channel is used to specify the effective low address for the MFC command, or the local storage pointer to the list elements for an MFC DMA list command. If translation is enabled in the MFC state register (that is, MFC_SR[R] is set to ‘1’), effective addresses are translated into real addresses by the address translation facility described in *Book III* of the *PowerPC Architecture*.

The contents of the MFC Effective Address Low or List Address Channel are not persistent and must be written for each MFC command enqueue sequence.

For transfer sizes less than 16 bytes, MFC_EAL bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be ‘0’. If translation is disabled, the MFC_EAL must be within the real address space limits of main storage domain.

For more information about the real address limits of an implementation, refer to the specific implementation documentation. For MFC list commands, bits 29 through 31 of the List Address must be ‘0’. If any of these conditions are not met, the MFC_EAL parameter is invalid and considered unaligned.

The validity of this parameter is checked asynchronous to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, a protection violation, or because the address is not aligned), MFC command queue processing is suspended and an interrupt is sent.

The types of interrupts that can be sent are:

- If a segment fault occurs, an MFC data-segment interrupt
- If a mapping fault or page protection violation occurs, an MFC data-storage interrupt
- If the address is not aligned, a DMA alignment interrupt

For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Note: The validity of the effective address is checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type Write

Channel Number x‘12’

Low Word of 64-bit Effective Address or the Local Storage Address of the MFC List

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	Low Word of 64-bit Effective Address or the Local Storage Address of the MFC List	Low word of the 64-bit effective address or the local storage address of the MFC list. For transfer sizes less than 16 bytes, address bits 28 through 31 must provide natural alignment based on the transfer size. For transfer sizes of 16 bytes or greater, bits 28 through 31 must be ‘0’. For optimal performance of transfers of 128 bytes or more, the source and destination transfer addresses should be 128-byte aligned (bits 25 through 31 set to ‘0’). If translation is disabled (the effective address equals the real address), some higher-order bits must also be ‘0’, depending on the amount of real memory in the system. For MFC list operations, this parameter contains the local storage address of the MFC list. In this case, the address must begin on an 8-byte boundary.

9.1.7 MFC Effective Address High Channel (MFC_EAH)

The MFC Effective Address High Channel is used to specify the effective address for the MFC command. If translation is enabled in the MFC state register (that is, MFC_SR1[R] is ‘1’), effective addresses are translated into real addresses by the address translation facility described in *Book III* of the *PowerPC Architecture*.

The contents of the MFC Effective Address High Channel are not persistent and must be written for each MFC command enqueue sequence.

If the upper 32 bits are not written, hardware sets EAH, the high address bits are set to zeros (that is, the address is between 0 and 4 GB).

The validity of this parameter is checked asynchronous to the instruction stream. If the address is invalid (for example, due to a segment fault, a mapping fault, or a protection violation), MFC command queue processing is suspended and an interrupt is sent.

The types of interrupts that can be sent are:

- If a segment fault occurs, an MFC data-segment interrupt
- If a mapping fault or page protection violation occurs, an MFC data-storage interrupt

For more information on interrupts, see *Section 21 Interrupt Facilities* beginning on page 237.

Note: The validity of the effective address is checked during transfers. Partial transfers can be performed before an invalid address is encountered and the exception is generated.

Access Type Write

Channel Number x‘11’

High Word of 64-bit Effective Address (Optional)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	High Word of 64-bit Effective Address	High word of the 64-bit effective address. Zeros are used for the upper 32 bits of the effective address if this parameter is omitted from the command. If translation is disabled (the effective address equals the real address), some higher-order bits must also be zeros, depending on the amount of real memory in the system.

9.2 MFC SPU Command Issue Sequence

To queue an MFC command from the SPU, the MFC command parameters must first be written to the MFC command parameter channels in any order, except that step 6 must always be done last:

1. Write the local storage address parameter (32 bits).
2. Write the effective address high parameter (upper 32 bits).[1]
3. Write the effective address low or the list address parameter (lower 32 bits).
4. Write the MFC transfer or list size parameter (16 bits).
5. Write the MFC command tag parameter (16 bits).
6. Write the MFC command opcode and class ID parameter (32 bits).[2]

The MFC command parameters are retained in the MFC command parameter channels until a write of the MFC command opcode and class ID parameter is processed by the MFC.

A write channel (**wrch**) to the MFC Command Opcode Channel and MFC Class ID Channel causes the parameters held in the MFC command parameter channels to be sent to the MFC command queue. The MFC command parameters can be written in any order before the issue of the MFC command itself. The values of the last parameters written to the command parameter channels are used in the enqueuing operation.

After an MFC command has been queued, the values of the MFC parameters become invalid and must be respecified for the next MFC command queuing request. Not specifying all of the required MFC parameters (that is, all the parameters except for the optional EAH) can result in the improper operation of the MFC command queue.

MFC command parameter channels are non-blocking, and do not have channel counts associated with them. A read channel count (**rchcnt**) instruction that is sent to these channels returns a count of ‘1’.

The MFC Command Opcode Channel and MFC Class ID Channel have a maximum count configured by hardware to the number of MFC queue commands supported by hardware. Software must initialize the channel count of the MFC Command Opcode Channel to the number of empty MFC proxy command queue slots supported by the implementation after power on and after a purge of the MFC proxy command queue. The channel count of the MFC Command Opcode Channel must also be saved and restored on an SPE preemptive context switch. (For more information, see *the SPE Context Save Sequence* in the specific implementation documentation.*)*

[1] This parameter is optional and is set to ‘0’ if not written.
[2] This write channel (**wrch**) command stalls the SPU until there is room in the MFC queue for the command.

9.3 MFC Tag-Group Status Channels

This section describes the procedures for determining the completion of MFC commands for a tag group. It also describes the procedures for determining if an MFC DMA list command is stalled on a list element, which has its stall-and-notify flag set to a ‘1’ and how to subsequently restart the MFC DMA list command.

Each command is tagged with a 5-bit identifier (that is, the MFC command tag). The same identifier can be used for multiple MFC commands. A set of commands with the same identifier is defined as a tag group. Software can use this identifier (that is, the MFC command tag) to check, or to wait on, the completion of all queued commands for each tag group. In addition, the MFC command tag is used by software to check, or to wait on, an MFC DMA list command, to reach an element with a stall-and-notify flag set, and to acknowledge the list element for resuming the MFC DMA list command. For more information, on DMA List Elements (see page 53).

9.3.1 Procedures for Determining the Status of Tag Groups

Three basic procedures are supported to determine the status of the tag groups:

- Polling the MFC Read Tag-Group Status Channel
- Waiting for a tag-group update, or waiting for an event
- Interrupting on a tag-group status update event

For polling for the completion of an MFC command or for the completion of a group of MFC commands, the basic procedure is as follows:

1. Clear any pending tag status update requests by:
    - Writing a ‘0’ to the MFC Write Tag Status Update Request Channel (see page 116)
    - Reading the channel count associated with the MFC Write Tag Status Update Request Channel (see page 116), until a value of ‘1’ is returned
    - Reading the MFC Read Tag-Group Status Channel (see page 117) and discarding the tag status data.
2. Enable the tag groups of interest by writing the MFC Write Tag-Group Query Mask Channel (see page 114) with the appropriate mask data (only needed if a new tag-group mask is required).
3. Request an immediate tag status update by writing the MFC Write Tag Status Update Request Channel (see page 116) with a value of ‘0’.
4. Perform a read of the MFC Read Tag-Group Status Channel (see page 117). The data returned is the current status of each tag group with the tag-group mask applied.[1]
5. Repeat steps 3 and 4 until the tag group or the tag groups of interest are complete.

Waiting for a tag-group update, or waiting for events (one or more tag-group completions) the basic procedure is as follows:

1. Clear any pending tag status update requests by:
    - Writing a ‘0’ to the MFC Write Tag Status Update Request Channel (see page 116)
    - Reading the channel count associated with the MFC Write Tag Status Update Request Channel (see page 116), until a value of ‘1’ is returned
    - Reading the MFC Read Tag-Group Status Channel (see page 117) and discarding the tag status data.

[1] Performing a read of the SPU Tag Group Status before issuing a request for a tag status update results in an SPU execution deadlock.

2. Request a conditional tag status update by writing the MFC Write Tag Status Update Request Channel (see page 116), with a value of '01' or '10'. A value of '01' specifies that the completion of *any* enabled tag group results in a tag-group update. A value of '10' specifies that *all* enabled tag groups must complete to result in an SPU tag-group status update.
3. THEN EITHER:
    - Read from the MFC Read Tag-Group Status Channel (see page 117) to wait on the specific tag event specified in steps 1 and 2. This read stalls the execution of the SPU until the condition as specified in step 2 is met.

OR

- Perform a read of the count associated with the MFC Read Tag-Group Status Channel (see page 117) to poll or wait for the specific tag event.

4. Repeat the previous step until the count is returned as a ‘1’.
5. Now, read from the MFC Read Tag-Group Status Channel (see page 117) to determine which tag group or tag groups are complete.

An alternative to waiting or to polling on a conditional tag event is to use the SPU event facility. This procedure is typically used when an application is waiting for one of multiple events to occur or can do other work while waiting for command completion. The procedure is as follows:

1. Clear any pending tag status update requests by:
    - Writing a ‘0’ to the MFC Write Tag Status Update Request Channel (see page 116)
    - Reading the channel count associated with the MFC Write Tag Status Update Request Channel (see page 116) until a value of ‘1’ is returned
    - Reading the MFC Read Tag-Group Status Channel (see page 117) and discarding the tag status data.

    After the above step, the tag group or tag groups are selected.
2. Clear any pending tag status update events by writing (**wrch**) the SPU Write Event Acknowledgment Channel (see page 144) with a value of ‘1’.
3. Unmask the MFC Tag-Group Status Update Event (see page 146) by writing a '1' to the SPU Write Event Mask Channel (see page 140).
4. THEN EITHER:
    - Read from the SPU Read Event Status Channel (see page 136) to wait for an enabled event to occur. This read stalls the execution of the SPU until an enabled event occurs.

OR

- Read the count associated with the SPU Read Event Status Channel (see page 136) to poll or wait for the specific tag event until the count is returned as a ‘1’.

5. Read from the SPU Read Event Status Channel (see page 136) to determine which events occurred.
6. If an MFC Tag-Group Status Update Event occurred, read (**rdch**) from the MFC Read Tag-Group Status Channel (see page 117) to determine which tag or tag groups caused the event.

9.3.2 Procedures for Determining MFC DMA List Command Completion

Three basic procedures are supported to determine if an MFC DMA list command has reached a list element with the stall-and-notify flag set:

- Poll the MFC Read List Stall-and-Notify Tag Status Channel (see page 118)
- Wait for a MFC DMA List Command Stall-and-Notify Event (see page 146)
- Interrupt on MFC DMA List Command Stall-and-Notify Event.

The basic procedure for polling to determine if an MFC DMA list command has reached a list element with the stall-and-notify flag set is as follows:

1. Issue an MFC DMA list command which has a list element with the stall-and-notify flag set.
2. Read (**rchcnt**) the count associated with the MFC Read List Stall-and-Notify Tag Status Channel (see page 118) until a value of ‘1’ is returned.
3. Perform a read (**rdch**) of the MFC Read List Stall-and-Notify Tag Status Channel (see page 118). The data returned is the current status of each tag group, which has reached a list element with the stall-and-notify flag set since the last read of this channel.
4. Repeat steps 2 and 3 until the tag group or tag groups of interest have reached the list element with the stall-and-notify flag set. (The corresponding bits are set in the return data.)
5. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119) with the tag-group number corresponding to the stalled tag group to resume the MFC DMA list command.

The basic procedure for waiting on an MFC DMA list command to reach a list element with the stall-and-notify flag set is as follows:

1. Issue an MFC DMA list command which have a list element with the stall-and-notify flag set.
2. Perform a read (**rdch**) of the MFC Read List Stall-and-Notify Tag Status Channel (see page 118). The data returned is the current status of each tag group which has reached a list element with the stall-and-notify flag set since the last read of this channel. This read stalls the SPU until an MFC DMA list command has reached a list element with the stall-and-notify flag set.
3. Repeat step 2 until the tag group or groups of interest have reached the list element with the stall-and-notify flag set. (The corresponding bits are set in the return data. Since the bits are reset for each read, software must perform the accumulation of the tag groups while waiting on multiple tag groups to stall.)
4. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119) with a '1' in the corresponding bit position to restart the MFC DMA list command.

An alternative to waiting or polling on List Stall-and-Notify Tag Group Status is to use the SPU event facility. This procedure is typically used when other work can be performed by the SPU program while the MFC DMA list command is executing.

The procedure is as follows:

1. Clear any pending MFC DMA List Command Stall-and-Notify Events by writing (**wrch**) the SPU Write Event Acknowledgment Channel (see page 144) with a value of ‘1’.
2. Enable the MFC DMA List Command Stall-and-Notify Event by writing a ‘1’ to the Sn bit of the SPU Write Event Mask Channel (see page 140).
3. Issue an MFC DMA list command which have a list element with the stall-and-notify flag set.

4. THEN EITHER:

    1. Read (**rdch**) from the SPU Read Event Status Channel (see page 136) to wait for an enabled event to occur. This read stalls the execution of the SPU until an enabled event occurs.

OR:

1. Read (**rchcnt**) the count associated with the SPU Read Event Status Channel (see page 136) to poll for the specific tag event until the count is returned as a ‘1’.
2. Read **(rdch**) from the SPU Read Event Status Channel (see page 136) to determine which events occurred.

5. If a DMA list stall-and-notify event occurred, read (**rdch**) from the MFC Read List Stall-and-Notify Tag Status Channel (see page 118) to determine which tag group or groups caused the event.
6. Write (**wrch**) the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119) with the tag-group number corresponding to the stalled tag group to resume the MFC DMA list command.

9.3.3 MFC Write Tag-Group Query Mask Channel (MFC_WrTagMask)

The MFC Write Tag-Group Query Mask Channel selects the tag groups to be included in the query or wait operations. The data provided by this channel is retained by the MFC until changed by a subsequent write channel (**wrch**) to this channel. Therefore, the data does not need to be respecified for each status query or wait. If this mask is modified by software when an MFC tag status update request is pending, the meaning of the results are ambiguous. A pending MFC tag status update request should always be cancelled before a modification of this mask. An MFC tag status update request can be cancelled by writing a value of ‘0’ (that is, immediate update) to the MFC Write Tag Status Update Request Channel (see page 116).

The current contents of this channel can be accessed by reading (**rdch**) the MFC Read Tag-Group Query Mask Channel (see page 115).

This channel is non-blocking and does not have an associated count. If a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Write

Channel Number x‘16’

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_F	g_E	g_D	g_C	g_B	g_A	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” select 0 Tag group is not part of a query or wait-on-tag-event operation. 1 Tag group is part of a query or wait-on-tag-event operation.

9.3.4 MFC Read Tag-Group Query Mask Channel (MFC_RdTagMask)

The MFC Read Tag-Group Query Mask Channel is used to read the current value of the Proxy Tag-Group Query Mask Register (see page 84) Reading this channel always returns the last data written to the MFC Write Tag-Group Query Mask Channel (see page 114).

The MFC Read Tag-Group Query Mask Channel allows software to read the state of the Proxy Tag-Group Query Mask Register. This channel can be used to avoid software shadow copies of the Proxy Tag-Group Query Mask for the SPE context save and restore operations.

This channel is non-blocking and does not have an associated count. If a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Read

Channel Number x‘C’

g1F	g1E	g1D	g1C	g1B	g1A	g19	g18	g17	g16	g15	g14	g13	g12	g11	g10	gF	gE	gD	gC	gB	gA	g9	g8	g7	g6	g5	g4	g3	g2	g1	g0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” select. 0 Tag group is not part of a query or wait-on-tag-event operation. 1 Tag group is part of a query or wait-on-tag-event operation.

9.3.5 MFC Write Tag Status Update Request Channel (MFC_WrTagUpdate)

The MFC Write Tag Status Update Request Channel controls when the MFC tag-group status is updated in the MFC Read Tag-Group Status Channel (see page 117).

The MFC Write Tag Status Update Request Channel can specify that the status be:

- Updated immediately
- Updated when *any* enabled MFC tag-group completion has a “no operation outstanding” status, or
- Updated only when *all* enabled MFC tag groups have a “no operation outstanding” status.

A write channel (**wrch**) instruction to this channel must occur before a read channel (**rdch**) from the MFC Read Tag-Group Status Channel occurs.

An MFC Write Tag Status Update request should be performed after setting the tag-group mask and after issuing the commands for the tag groups of interest. If the commands for a tag group are completed before issuing the MFC Write Tag Status Update request, thereby satisfying the update status condition, the status is returned without waiting.

Reading from the MFC Read Tag-Group Status Channel (see page 117) without first requesting a status update by writing to the MFC Write Tag Status Update Request Channel results in a software-induced deadlock.

A previous MFC Tag Status Update Request can be cancelled by completing the following steps:

1. Issue an immediate update status request to the MFC Write Tag Status Update Request Channel.
2. Read the count associated with the MFC Write Tag Status Update Request Channel until a value of ‘1’ is returned.
3. Read from the MFC Read Tag-Group Status Channel to determine and to discard unwanted results.

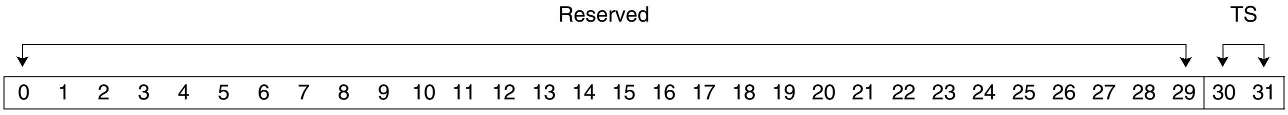
Two conditional update requests without an intervening status read request result in the return of an unpredictable tag status. To avoid the unpredictable results, software should pair requests for tag status updates with reads of the tag status, unless a request cancellation is being performed via the immediate-update request.

Privileged software initializes the count for this channel to ‘1’. The count for this channel is set to ‘0’ when a write channel (**wrch**) instruction is issued to this channel. The count is set to ‘1’ when the MFC receives the tag status update request.

This channel is write-blocking enabled with a maximum count of ‘1’.

Access Type Write-blocking

Channel Number x‘17’



Bits	Field Name	Description
0:29	Reserved	Reserved.

Bits	Field Name	Description
30:31	TS	Tag-status update condition 00 Update immediately, unconditional. 01 Update tag status if or when *any* enabled tag group has “no outstanding operation” status. 10 Update tag status if or when *all* enabled tag groups have “no outstanding operation” status. 11 Reserved.

Implementation Note:

When an immediate request type command is issued to this channel, the MFC must not acknowledge the write channel (**wrch**) of the MFC Write Tag Status Update Request Channel until it has processed the command and updated the MFC Read Tag-Group Status Channel with the results.
An immediate request type command must cancel any pending conditional request command. When the MFC has advised the hardware to check for the condition, the MFC acknowledges the write channel (**wrch**) of the MFC Write Tag Status Update Request Channel.
When a conditional request type command is issued to this channel, acknowledging the write of this channel causes the channel count to change from ‘0’ to ‘1’.

9.3.6 MFC Read Tag-Group Status Channel (MFC_RdTagStat)

The MFC Read Tag-Group Status Channel contains the status of the tag groups from the last tag-group status update request. Only the status of the enabled tag groups at the time of the tag-group status update is valid. The bit positions that correspond to the tag groups that are disabled at the time of the tag-group status update are set to '0'.

An MFC Write Tag Status Update Request Channel must be requested before reading from this channel. Failure to do so results in a software-induced deadlock condition. This is considered a programming error, and privileged software is required to remove the deadlock condition.

A read channel count (**rchcnt**) instruction sent to the MFC Read Tag-Group Status Channel returns ‘0’ if the status is not yet available or returns ‘1’ if the status is available. This instruction can be used to avoid stalling the SPU when the MFC Read Tag-Group Status Channel is read.

Software initializes the count for this channel to a value of ‘0’. This channel is read-blocking enabled, with a maximum count of ‘1’.

Access Type Read-blocking

Channel Number x‘18’

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_F	g_E	g_D	g_C	g_B	g_A	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” status 1 Tag group has no outstanding operations (commands are complete), and was not disabled by the query mask. 0 Tag group has outstanding operations, or has been disabled by the query mask.

9.3.7 MFC Read List Stall-and-Notify Tag Status Channel (MFC_RdListStallStat)

List elements for an MFC list command contain a stall-and-notify flag. If the flag is set on a list element, the MFC stops executing MFC list command and sets the bit corresponding to the tag group of the MFC list command in this channel. The count associated with this channel is also set to ‘1’. An MFC list command remains stalled until acknowledged by writing the tag value to the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119).

Note: The MFC List Stall and Notify facility is useful when a program wishes to be notified when a DMA list execution has reached a specific point. This is also useful, when an application wishes to dynamically change list elements (transfer sizes or effective addresses) that follow the stalled list element. List elements can also be skipped by setting their transfer size to 0. Hardware is not allowed to prefetch list elements beyond a stall and notify element.

Privileged software should initialize the count of the MFC Read List Stall-and-Notify Tag Status Channel to zeros.

Software can determine which tag groups have commands that have stalled since the last read of this channel by reading the contents of this channel again. Issuing a read channel (**rdch**) instruction to this channel resets all bits to ‘0’s, and sets the count corresponding to this channel to a ‘0’; therefore, issuing a read channel (**rdch**) instruction with no outstanding list elements that contain a stall-and-notify flag set to '1' and no stalled commands, results in a software induced deadlock.

Issuing a read channel (**rdch**) instruction when no tag groups are stalled results in SPU execution stall until a list element with the stall-and-notify flag set is encountered.

Software can also read (**rchcnt**) the count associated with this channel or use the SPU event facility to determine when an MFC list element is encountered with the stall-and-notify flag set.

A read channel count (**rchcnt**) instruction sent to the MFC Read List Stall-and-Notify Tag Status Channel returns ‘0’ if there are no new stalled MFC list commands since the last read of this channel.

This channel is read-blocking and has a maximum count of ‘1’.

Access Type Read-blocking

Channel Number x‘19’

g_{1F}	g_{1E}	g_{1D}	g_{1C}	g_{1B}	g_{1A}	g_{19}	g_{18}	g_{17}	g_{16}	g_{15}	g_{14}	g_{13}	g_{12}	g_{11}	g_{10}	g_F	g_E	g_D	g_C	g_B	g_A	g_9	g_8	g_7	g_6	g_5	g_4	g_3	g_2	g_1	g_0
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	g_n	Tag group “n” status 1 Tag group has an MFC list command that has stalled on an element with the stall-and-notify flag set. 0 Tag group has no MFC list commands currently stalled.

Programming Note:

Programmers should avoid issuing multiple MFC DMA list commands, which have the stall and notify flags set within the same tag group, unless subsequent commands have a tag-specific fence. If multiple MFC DMA list commands are issued with the same tag id and without a tag-specific fence, software cannot determine which command or commands have reached the stall point. A programmer should always use a tag-specific fence, or a barrier between MFC DMA list commands with the same tag id and which have a list element with a stall and notify flag set.

Implementation Note:

1. If setting a bit occurs at the same time as a read channel (**rdch**) instruction, which resets all bits, that bit must remain set and the channel count must remain at ‘1’, if the state of the bit is not reflected in the data returned for the read channel (**rdch**) instruction.
2. If MFC list prefetching is implemented by hardware, it must not span an element that has the stall-and-notify flag set. Software can modify the list following a stall, before acknowledging and resuming list processing.

9.3.8 MFC Write List Stall-and-Notify Tag Acknowledgment Channel (MFC_WrListStallAck)

The MFC Write List Stall-and-Notify Tag Acknowledgment Channel is used to acknowledge a tag group containing MFC list commands that are stalled on a list element with the stall-and-notify flag set. The tag group is acknowledged by writing the MFC tag group to this channel. After the write, all stalled MFC list commands of the tag group, which matches the value written to this channel are restarted.

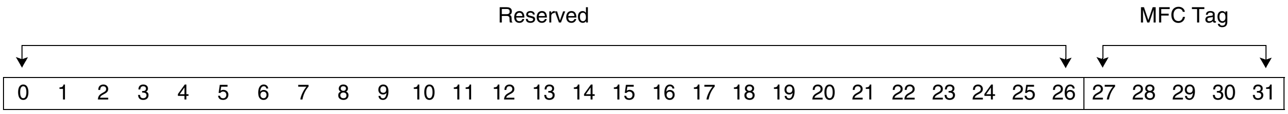
Note: The MFC List Stall and Notify facility is useful when a program wishes to be notified when a DMA list execution has reached a specific point. This is also useful, when an application wishes to dynamically change list elements (transfer sizes or effective addresses) that follow the stalled list element. List elements can also be skipped by setting their transfer size to 0. Hardware is not allowed to prefetch list elements beyond a stall and notify element.

Acknowledging a tag group that is currently not stalled due to a stall-and-notify condition is undefined. Doing so results in an invalid status in the MFC Read List Stall-and-Notify Tag Status Channel. For consistency, an implementation should treat this condition as a no-op.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Write

Channel Number x‘1A’



Bits	Field Name	Description
0:26	Reserved	Reserved.
27:31	MFC Tag	The tag can be any value between x‘0’ and x‘1F’.

9.4 MFC Read Atomic Command Status Channel (MFC_RdAtomicStat)

The MFC Read Atomic Command Status Channel contains the status of the last completed immediate MFC atomic update command (**getllar, putllc**, or **putlluc**). Issuing a channel read (**rdch**) instruction to this channel before issuing an immediate atomic command results in a software-induced deadlock.

Note: This channel does not provide any status for the queued **putqlluc** command.

Software can read the channel count (**rchcnt**) associated with this channel to determine if an immediate atomic MFC command has completed.

- If a value of ‘0’ is returned, the immediate atomic MFC command has not completed.
- If a value of ‘1’ is returned, the immediate atomic MFC command has completed and the status is available by reading (**rdch**) this channel.

A read (**rdch**) from the MFC Read Atomic Command Status Channel should always follow an immediate atomic MFC command. Performing multiple atomic MFC commands without an intervening read from the MFC Read Atomic Command Status Channel results in an incorrect status.

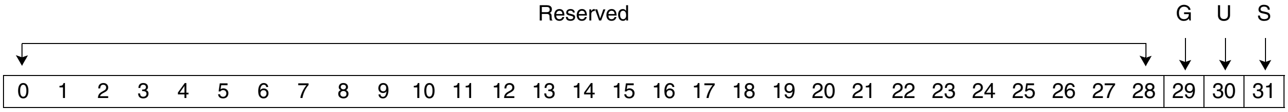
Privileged software should initialize the count of this channel to ‘0’. This channel is read-blocking with a maximum count of ‘1’.

The contents of this channel are cleared when read.

Completion of a subsequent immediate MFC atomic update commands overwrites the status of earlier MFC commands.

Access Type Read-blocking

Channel Number x‘1B’



Bits	Field Name	Description
0:28	Reserved	Reserved.
29	G	Set if the get lock-line and reserve (**getllar**) command completed.
30	U	Set if the put lock-line unconditional (**putlluc**) command completed.
31	S	Put lock-line conditional command (**putllc**). 1 Put conditional unsuccessful. The reservation was lost. 0 Put conditional successful.

9.5 SPU Mailbox Channels

This section contains a description of:

- The SPU Write Outbound Mailbox Channel (see page 122)
- The SPU Write Outbound Interrupt Mailbox Channel (see page 123)
- The SPU Read Inbound Mailbox Channel (see page 124).

These channels are all part of the Mailbox Facility (see page 90).

The SPU mailbox channels are defined as blocking (that is, they stall the SPU when a channel is full, write-blocking, or when data are not available, read-blocking). The blocking method of a channel is very beneficial for power savings when an application has no other work to perform.

However, accessing these channels causes the SPU to stall for an indefinite period of time. Software can avoid stalling the SPU by using the SPU Event Facility (see page 133) or by reading the channel count associated with the mailbox channel.

9.5.1 SPU Write Outbound Mailbox Channel (SPU_WrOutMbox)

A write channel (**wrch**) to this channel writes data to the SPU write outbound mailbox queue. The data written to this channel by the SPU is available for an MMIO read of the SPU Outbound Mailbox Register (see page 91).

A write channel (**wrch**) to this channel also causes the associated channel count to be decremented by ‘1’. Writing to a full SPU write outbound mailbox queue causes SPU execution to stall until the SPU Outbound Mailbox Register is read, freeing up a location in the SPU write outbound mailbox queue.

To avoid the stall condition, the channel count associated with this channel can be read to ensure there is a slot in the SPU write outbound mailbox queue before issuing the channel write. Alternatively, the SPU Outbound Mailbox Available Event can be used to signal the availability of a slot in the SPU write outbound mailbox queue, if it was determined to be full.

When the SPU write outbound mailbox queue is full, a read of the channel count associated with this channel returns a value of ‘0’; a non-zero value indicates the number of 32-bit words free in the SPU write outbound mailbox queue.

Privileged software initializes the count of the SPU Write Outbound Mailbox Channel to the depth of the SPU write outbound mailbox queue. This channel is write-blocking. The maximum count for this channel is implementation-dependent and should be the depth (that is, the number of available slots) of the SPU write outbound mailbox queue.

Access Type Write-blocking

Channel Number x‘1C’

Mailbox Data

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC does not acknowledge a write channel (**wrch**) instruction to the SPU Write Outbound Mailbox Channel until a processor or device has read the contents of the mailbox.

9.5.2 SPU Write Outbound Interrupt Mailbox Channel (SPU_WrOutIntrMbox)

A write channel (**wrch**) instruction to this channel writes data to the SPU write outbound interrupt mailbox queue. The data written to this channel by the SPU is made available to an MMIO read of the SPU Outbound Interrupt Mailbox Register (see page 213). (This register is located in the Privilege 2 area of the SPE main storage address domain.)

A write channel (**wrch**) instruction to this SPU Write Outbound Mailbox Channel also causes the associated count to be decremented by ‘1’. Writing to a full SPU write outbound interrupt mailbox queue causes SPU execution to stall until the SPU Outbound Interrupt Mailbox Register (see page 213) is read, freeing up a location in the SPU write outbound interrupt mailbox queue.

To avoid a stall condition, the channel count associated with this channel can be read to ensure there is a slot in the SPU write outbound interrupt mailbox queue before issuing the channel write. Alternatively, the SPU Outbound Interrupt Mailbox Available Event can be used to signal the availability of a slot in the SPU write outbound interrupt mailbox queue, if it was previously full.

A write channel (**wrch**) instruction to the SPU Write Outbound Interrupt Mailbox Channel also causes an interrupt to be sent to a processor or other device. There is no ordering of the interrupt and previously-issued MFC commands.

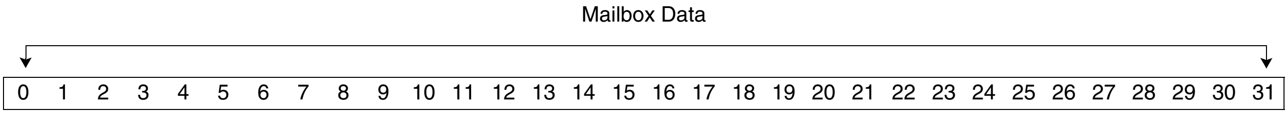
For more information, see *Section 21 Interrupt Facilities* beginning on page 237.

When the SPU write outbound interrupt mailbox queue is full, a read of the count associated with this channel returns a value of ‘0’; the non-zero count value has the number of 32-bit words free in this queue.

Privileged software initializes the count of this channel to the depth of the SPU write outbound interrupt mailbox queue. This channel is write-blocking. The maximum count for this channel is implementation-dependent and should be the depth (that is, the number of available slots) of the SPU write outbound interrupt mailbox queue.

Access Type Write-blocking

Channel Number x‘1E’



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge the write of the SPU Write Outbound Interrupt Mailbox Channel until the processor or other devices have read the contents of the mailbox.

9.5.3 SPU Read Inbound Mailbox Channel (SPU_RdInMbox)

A read from this channel returns the next data in the SPU read inbound mailbox queue. Data is placed in the SPU read inbound mailbox queue by a processor or device issuing write to the SPU Inbound Mailbox Register (see page 92). Reading from the SPU Read Inbound Mailbox Channel causes the associated count to be decremented by ‘1’.

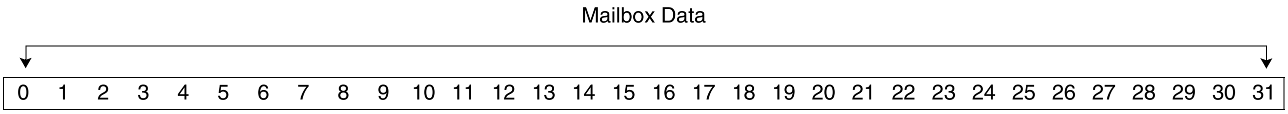
Reading an empty mailbox causes SPU execution to stall until the SPU Inbound Mailbox Register (see page 92) is written, placing a data item in the SPU read inbound mailbox queue. To avoid the stall condition, the channel count associated with this channel can be read to ensure there is data in the SPU read inbound mailbox queue before issuing the channel read. Alternatively, the SPU Inbound Mailbox Available Event can be used to signal the availability of data in the SPU read inbound mailbox queue.

If the mailbox is empty, reading the channel count (**rchcnt**) returns a value of ‘0’. If the result of the **rchcnt** is non-zero, then the mailbox contains information, which has been written by the PPE, but which has not been read by the SPU.

The channel count of the SPU Read Inbound Mailbox Channel is initialized by privileged software to ‘0’. The maximum count is implementation-dependent. This channel is read-blocking.

Access Type Read-blocking

Channel Number x‘1D’



Bits	Field Name	Description
0:31	Mailbox Data	Application-specific mailbox data Each application can uniquely define the mailbox data.

9.6 SPU Signalling Channels

These channels are the PPE part of the SPU Signal Notification Facility (see page 94). They are used to read signals from other processors and other devices in the system. The signalling channels are configured as read-blocking with a maximum count of ‘1’. When a read channel (**rdch**) instruction to one of these channels and the associated channel count is ‘1’, the current contents of the channel and the associated count are reset to ‘0’.

When a read channel (**rdch**) instruction to one of these channels, and the channel count is ‘0’, the SPU stalls until a processor or device performs an MMIO write to the associated register.

Implementation Note:

When a signalling event coincides with a read of the signalling channel, hardware must ensure that the proper state of the channel is maintained. In overwrite mode, hardware can either return the data associated with the signalling event or the current contents of the channel (if the count is ‘1’) for the channel read instruction. For logical OR mode, hardware can either return the data associated with the signalling event ORed with the current contents of the channel or the current contents of the channel. If the signalling event data or the logical OR of the event data and the current contents is returned, the channel count must end up as a ‘0’. If the current contents are returned, the channel count must end up as ‘1’.

9.6.1 SPU Signal Notification 1 Channel (SPU_RdSigNotify1)

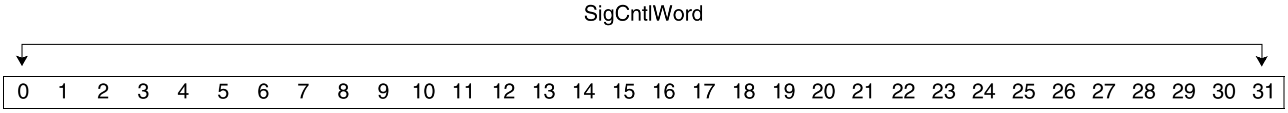
A read channel (**rdch**) instruction sent to the SPU Signal Notification 1 Channel returns the 32-bit value of signal-control word 1, and atomically resets any bits that were set when read. If no signals are pending, a read from this channel stalls the SPU until a signal is issued.

If no signals are pending, a read channel count (**rchcnt**) instruction sent to this channel returns ‘0’. If unread signals are pending, it returns ‘1’.

Privileged software initializes the count for this channel to a value of ‘0’. This channel is read-blocking enabled with a maximum count of ‘1’.

Access Type Read-blocking

Channel Number x‘3’



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word. The data is defined by the application. It can either be ORed with the previous value, or it can be overwritten.

9.6.2 SPU Signal Notification 2 Channel (SPU_RdSigNotify2)

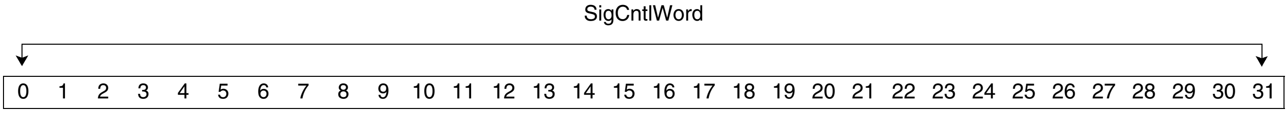
A read channel (**rdch**) instruction sent to the SPU Signal Notification 2 Channel returns the 32-bit value of signal-control word 2, and atomically resets any bits that were set when read. If no signals are pending, a read from this channel stalls the SPU until a signal is issued.

A read channel count (**rchcnt**) instruction sent to this channel returns ‘0’ if no signals are pending. It returns ‘1’ if unread signals are pending.

Privileged software initializes the count for this channel to a value of ‘0’. This channel is read-blocking enabled with a maximum count of ‘1’.

Access Type Read-blocking

Channel Number x‘4’



Bits	Field Name	Description
0:31	SigCntlWord	Signal-control word. The data is defined by the application. It can either be ORed with the previous value, or it can be overwritten.

9.7 SPU Decrementer

Each SPU contains a 32-bit decrementer. If enabled in the MFC Control Register (see page 209), it is written with MFC_CNTL[Dh] set to '0'. The SPU decrementer starts when a **wrch** instruction is issued to the SPU Write Decrementer Channel. The decrementer is stopped by following the program sequence described in *Section 9.7.1*, or when the MFC Control Register is written with MFC_CNTL[Dh] set to '1'.

The current running status of the decrementer is available in the MFC Control Register (that is, MFC_CNTL[Ds]). A decrementer event does not need to be pending for the decrementer to be stopped.

Note: The requirement for a constant rate for the time base and the decrementer is an additional requirement beyond the PowerPC Architecture. However, it is a requirement that all SPU decrementer run at the same rate as the PPE decrementer, but there is no requirement that the decrementers be synchronized.

Two channels are assigned to manage the decrementer: one to set the decrementer value and one to read the current contents of the decrementer. A decrementer event occurs when the most significant bit (bit 0) changes from a ‘0’ to a ‘1’.

9.7.1 SPU Write Decrementer Channel (SPU_WrDec)

The SPU Write Decrementer Channel is used to load a 32-bit value to the decrementer. The value loaded into the decrementer determines the lapsed time between the write channel (**wrch**) instruction and the decrementer event. The event occurs when the most significant bit of the decrementer changes from a ‘0’ to a ‘1’. If the value loaded into the decrementer causes a change from ‘0’ to ‘1’ in the MSb, an event is signaled immediately. Setting the decrementer to a value of ‘0’ results in an event after a single decrementer interval.

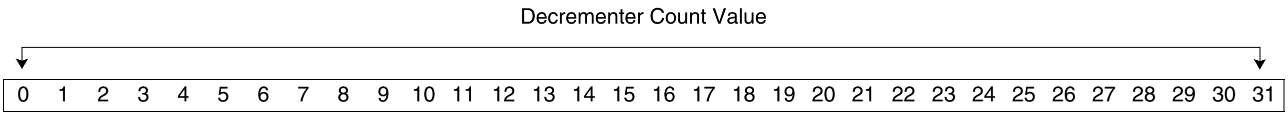
In order for the state of the decrementer to be properly saved and restored, the decrementer must be stopped before changing the decrementer value. The following sequence outlines the procedure for setting a new decrementer value.

1. Write to SPU Write Event Mask Channel to disabled decrementer event.
2. Write to SPU Write Event Acknowledgment Channel to acknowledge any pending events and to *stop the decrementer*. The decrementer is stopped because the decrementer event has been disabled in step 1.
3. Write to SPU Write Decrementer Channel to set a new decrementer count value. (Note: The decrementer is started because step 2 stopped the decrementer.)
4. Write to SPU Write Event Mask Channel to enable decrementer event.
5. Wait on timer to expire.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Write

Channel Number x‘7’



Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

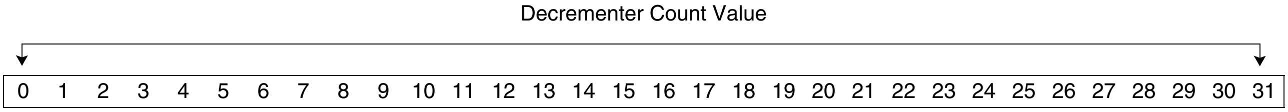
9.7.2 SPU Read Decrementer Channel (SPU_RdDec)

The SPU Read Decrementer Channel is used to read the current value of the 32-bit decrementer. Reading the decrementer count has no effect on the accuracy of the decrementer. Successive reads of the decrementer return the same value.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Read

Channel Number x‘8’



Bits	Field Name	Description
0:31	Decrementer Count Value	Decrementer count value.

9.8 SPU Read Machine Status Channel (SPU_RdMachStat)

The SPU Read Machine Status Channel contains the current SPU machine status information.This channel contains the two status bits: the isolation status and the SPU interrupt status. This isolation status reflects the current operating state of the SPU, isolated or nonisolated. For more on SPU isolation, see *Section 11 SPU Isolation Facility* beginning on page 163.

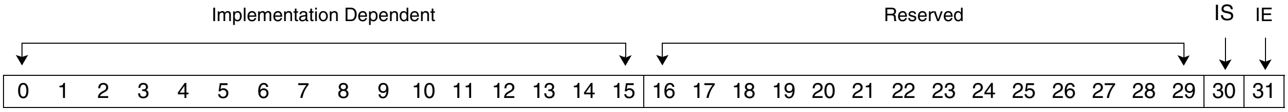
The SPU interrupt enable status reflects the current state of the SPU interrupt enable. If enabled, an SPU interrupt is generated if any enabled SPU event is present.For more information on SPU events, see *Section 9.11 SPU Event Facility* on page 133.

For more information on the processing of SPU interrupts, see the latest version of the *Synergistic Processor Unit Instruction Set Architecture* document listed in Related Publications (see page 16).

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Read

Channel Number x‘D’



Bits	Field Name	Description
0:15	Implementation Dependent	
16:29	Reserved	Set to zeros.
30	IS	Isolation status. 0 The SPU is operating in a non-isolated state. 1 The SPU is operating in an isolated state.
31	IE	SPU interrupt enable status. Interrupts can be enabled by setting the SPU_NPC[IE] bit to '1' while the SPU is stopped or by an SPU instruction. See the *Synergistic Processor Unit Instruction Set Architecture* for more information on how to enable interrupts. 0 SPU interrupts disabled. 1 SPU interrupt enabled.

9.9 SPU Interrupt-Related Channels

9.9.1 SPU Write State Save-and-Restore Channel (SPU_WrSRR0)

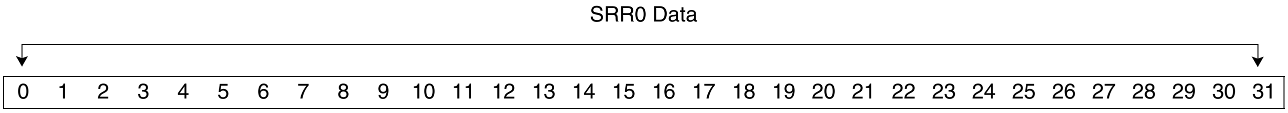
A write to this channel updates the contents of the state save and restore register (SRR0) in the SPU (for more information, see the *Synergistic Processor Unit Instruction Set Architecture*, listed in *Related Publications* as defined on page 16). A write to this channel is typically used to restore interrupt-state information when nested interrupts are supported.

This channel should not be written when SPU interrupts are enabled. Doing so can result in the contents of SRR0 being indeterminate. A channel form of the **sync** instruction (with the C bit in the instruction set) must be issued after writing this channel and before the execution of instructions that are dependent upon the SRR0 contents.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Write

Channel Number x‘E’



Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

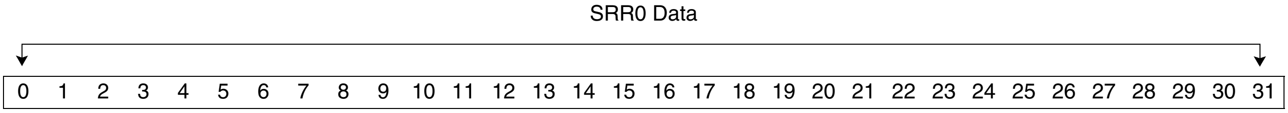
9.9.2 SPU Read State Save-and-Restore Channel (SPU_RdSRR0)

A read of this channel returns the contents of the state save and restore register (SRR0) in the SPU. A read of this channel is typically used to save interrupt-state information when nested interrupts are supported. (For more information, see the *Synergistic Processor Unit Instruction Set Architecture* document, listed in Related Publications (see page 16).)

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Read

Channel Number x‘F’



Bits	Field Name	Description
0:31	SRR0 Data	State save/restore register 0 data.

9.10 MFC Write Multisource Synchronization Request Channel (MFC_WrMSSyncReq)

The MFC Write Multisource Synchronization Request Channel is part of the MFC Multisource Synchronization Facility (see page 96) and causes the MFC to start tracking outstanding transfers sent to the associated MFC. When the synchronization requested by a write of this channel is complete, the channel count is set back to ‘1’. The data written to this channel is ignored; however, software should write a value of ‘0’ for compatibility with future enhancements.

A second write to this channel results in the SPU being stalled until the outstanding transfers being tracked by the first write are complete.

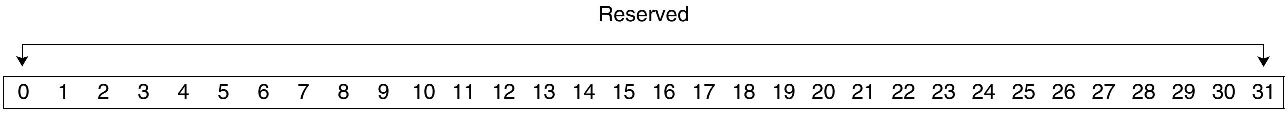
To use the MFC Write Multisource Synchronization Request Channel, a program must:

1. Write to the MFC Write Multisource Synchronization Request Channel.
2. Wait for the MFC Write Multisource Synchronization Request Channel to become available (that is, when the channel count is set back to ‘1’).

Software initializes the count for this channel to a value of ‘1’. This channel is write-blocking enabled with a maximum count of ‘1’.

Access Type Write-blocking

Channel Number x‘9’



Bits	Field Name	Description
0:31	Reserved	Reserved.

9.11 SPU Event Facility

An SPU program can monitor events by using:

- SPU Read Event Status Channel (see page 136)
- SPU Write Event Mask Channel (see page 140)
- SPU Read Event Mask Channel (see page 142)
- SPU Write Event Acknowledgment Channel (see page 144)

The SPU Read Event Status Channel contains the status of all events, which are enabled in the SPU Write Event Mask Channel. The SPU Write Event Acknowledgment Channel is used to reset the status of an event, usually an indication that the event has been processed, or recorded by the SPU program.

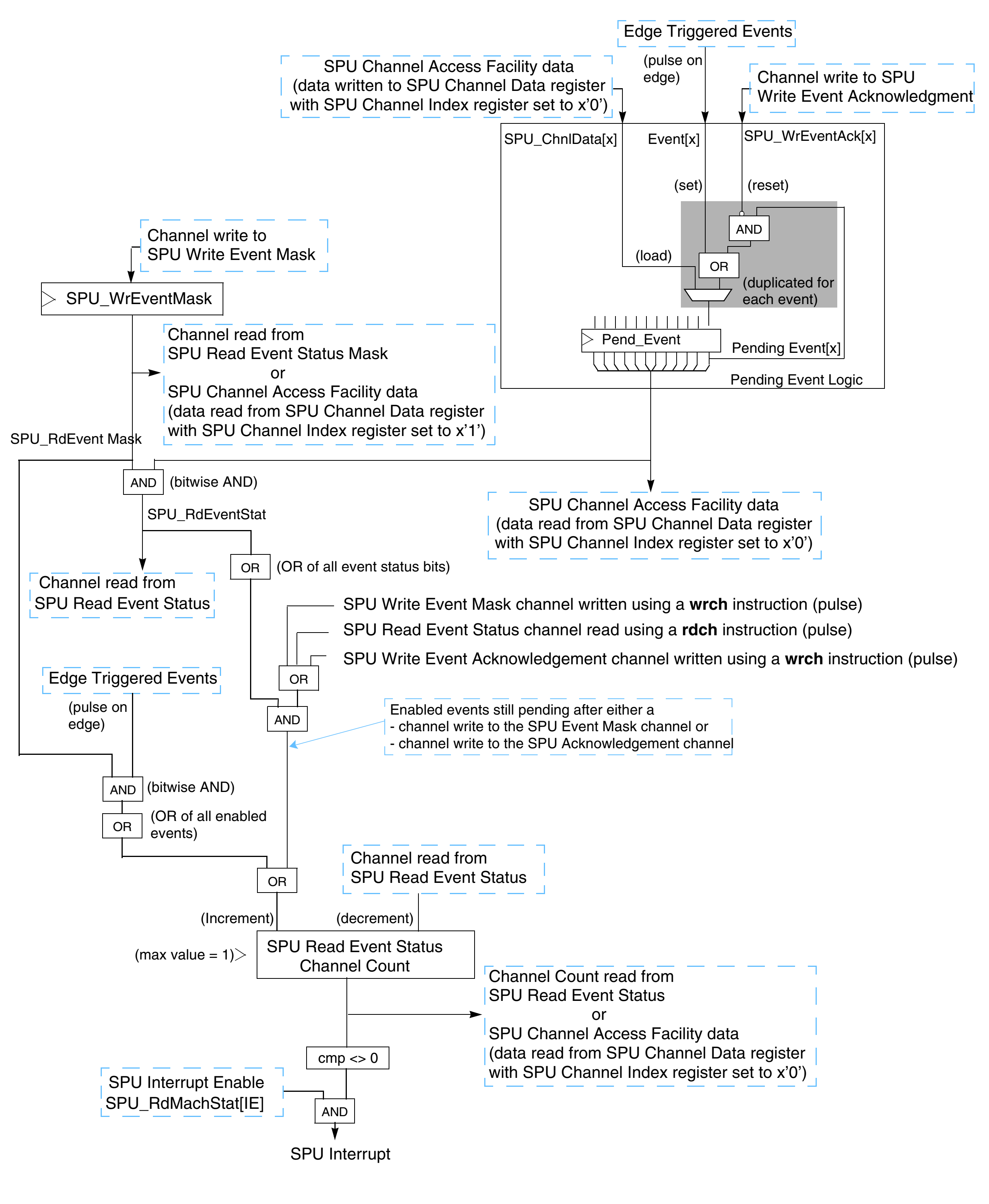
If no enabled events are present, reading from the SPU Read Event Status Channel stalls the SPU program.

While individual events have a similar methods for stalling the SPU program, if the event has not occurred, the SPU Event Facility provides software with a method to look for multiple events and to cause an interrupt of the SPU program.

There are several events which can be monitored. For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.

Figure 9-1 is a logical representation of the function of each channel that supports the SPU Event Facility.

Figure 9-1. Logical Representation of SPU Event Support


Edge Triggered Events
(pulse on edge)
SPU Channel Access Facility data (data written to SPU Channel Data register with SPU Channel Index register set to x'0')
Channel write to SPU Write Event Acknowledgment
SPU_ChnlData[x]
Event[x]
SPU_WrEventAck[x]
(set)
(reset)
AND
(load)
OR
(duplicated for each event)
Pend_Event
Pending Event[x]
Pending Event Logic
Channel write to SPU Write Event Mask
SPU_WrEventMask
Channel read from SPU Read Event Status Mask or SPU Channel Access Facility data (data read from SPU Channel Data register with SPU Channel Index register set to x'1')
SPU_RdEvent Mask
AND
(bitwise AND)
SPU_RdEventStat
SPU Channel Access Facility data (data read from SPU Channel Data register with SPU Channel Index register set to x'0')
OR
(OR of all event status bits)
Channel read from SPU Read Event Status
SPU Write Event Mask channel written using a **wrch** instruction (pulse)
SPU Read Event Status channel read using a **rdch** instruction (pulse)
SPU Write Event Acknowledgement channel written using a **wrch** instruction (pulse)
Edge Triggered Events
(pulse on edge)
OR
AND
Enabled events still pending after either a
- channel write to the SPU Event Mask channel or
- channel write to the SPU Acknowledgement channel
AND
(bitwise AND)
OR
(OR of all enabled events)
OR
Channel read from SPU Read Event Status
(Increment)
(decrement)
(max value = 1)
SPU Read Event Status Channel Count
Channel Count read from SPU Read Event Status or SPU Channel Access Facility data (data read from SPU Channel Data register with SPU Channel Index register set to x'0')
cmp <> 0
SPU Interrupt Enable SPU_RdMachStat[IE]
AND
SPU Interrupt

As illustrated in *Figure 9-1* on page 134, an edge-triggered event sets a corresponding bit in the SPU Pending Event Register to a ‘1’. Events in the SPU Pending Event Register are acknowledged, or reset, by writing a ‘1’ to the corresponding bit in the SPU Write Event Acknowledgment Channel (see page 144) using a channel instruction.

The SPU Pending Event Register (Pend_Event) is an internal register. The format of the SPU Pending Event Register is the same as the SPU Read Event Status Channel (see page 136). The SPU Pending Event Register can be read using the SPU Channel Access Facility (see page 218). Reading the SPU Read Event Status Channel with the channel read (**rdch**) instruction returns the value of the SPU Pending Event Register logically ANDed with the value in the SPU Write Event Mask Channel (see page 140). This function provides an SPU program with only the status of the enabled events, while the SPU Pending Event Register allows privileged software to see all the events which have occurred. Access to all events is required for an SPE context save and restore operation.

The contents of the SPU Read Event Status Channel (see page 136) change when the SPU Write Event Mask Channel is written with a new value, or when a new event is recorded in the SPU Pending Event Register. Any changing of a bit from ‘0’ to ‘1’ in the SPU Read Event Status Channel increments the SPU Read Event Status Channel count by ‘1’. The count also increments if an event is still set in the SPU Read Event Status Channel after a write to the SPU Write Event Acknowledgment Channel (see page 144). The count is decremented by ‘1’ when the SPU Read Event Status Channel is read using a channel read (**rdch**) instruction. The count saturates at a value of ‘1’, and is not decremented below a value of ‘0’.

When the SPU Read Event Status Channel count is nonzero, an interrupt condition is sent to the SPU, if enabled. Enabling, disabling, and processing of an interrupt are described in the *Synergistic Processor Unit Instruction Set Architecture* document. The most current version of this document is shown in the section, Related Publications (see page 16).

Programming Note:

Software should acknowledge all events to be processed before processing the events. For example, a pending SPU Inbound Mailbox Available Event should be acknowledged before reading the SPU Read Inbound Mailbox Channel. If an SPU Inbound Mailbox Available Event and an SPU Signal Notification 1 Available Event are to be processed, both events should be acknowledged in the same write to the SPU Write Event Acknowledgment Channel before reading the SPU Signal Notification 1 Channel. If each event were to be separately acknowledged, the event status count would be incremented unnecessarily and a phantom interrupt would be sent, if interrupts are enabled (that is, SPU_ RdMachStat[IE]).

9.11.1 SPU Read Event Status Channel (SPU_RdEventStat)

The SPU Read Event Status Channel contains the current status of all events enabled by the SPU Write Event Mask Channel (see page 140) at the time this channel is read. If the SPU Write Event Mask Channel specifies that an event is not part of the query, then its corresponding position is ‘0’ in the reported status.

A read from the SPU Read Event Status Channel, which has a channel count of ‘0’, results in an SPU stall; thereby providing a “wait on event” function. A read from this channel, with a channel count of ‘1’, returns the status of any enabled, pending events and set the channel count to ‘0’. The channel count is set to ‘1’ for the following conditions:

- An event occurs and the corresponding mask is '1' in the SPU Write Event Mask Channel (see page 140)
- The SPU Write Event Mask Channel is written with a '1' in a bit position which corresponds to a '1' in the SPU Pending Event Register
- Enabled events are pending after a write of the SPU Write Event Acknowledgment Channel (see page 144)
- Privileged software sets the channel count to ‘1’ using the SPU Channel Access Facility (see page 218).

If no enabled events have occurred, a read channel count (**rchcnt**) instruction of the SPU Read Event Status Channel returns zeros. A read channel count (**rchcnt**) instruction can be used to avoid stalling the SPU when reading the event status from the SPU Read Event Status Channel. The channel count of the SPU Read Event Status Channel is also used as the condition in the ‘branch indirect and set link if an external data’ **(bisled**) instruction. If the SPU Read Event Status Channel count is ‘0’, the branch is not taken. For more information on the **bisled** instruction, see the *Synergistic Processor Unit Instruction Set Architecture* document.

Privileged software must initialize the count value of the SPU Read Event Status Channel to ‘0’.[1] The channel count is initialized using the SPU Channel Count Register in the SPU Channel Access Facility (see page 218).

If SPU interrupts are enabled (SPU_ RdMachStat[IE] set to '1'), a non-zero SPU Read Event Status Channel count results in an interrupt being issued to the SPU.

Programming Notes:

Software can cause phantom events in two instances:

1. If software acknowledges or masks an event after the event has incremented the SPU Read Event Status Channel count, before reading the event status from the SPU Read Event Status Channel. In this case, reading the SPU Read Event Status Channel returns data that indicates that the event is no longer present or is disabled.
2. If software resets the interrupting condition of an enabled event (such as reading from a mailbox) before reading the SPU Read Event Status Channel and before acknowledging the event. In this case, reading the event-status register returns data that indicates that the event is still pending, even though the condition that generated the event is no longer present. In this case, the event must still be acknowledged.

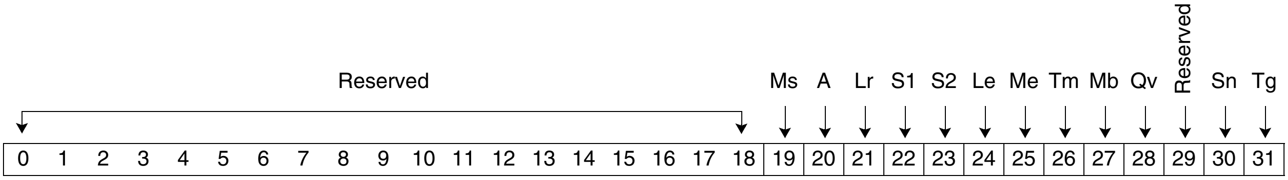
To avoid generating phantom events, events should be handled as follows:

- Read the SPU Read Event Status Channel.

[1] The SPU Read Event Status Channel has a depth of one. Therefore, its count can be only ‘0’ or ‘1’. The count is set to ‘1’ by Privileged software when restoring an SPE context with enabled, pending events.

- For all events that are to be processed, acknowledge the event by writing the corresponding bits to the SPU Write Event Acknowledgment Channel (see page 144).
- Process the events (for example, read the mailbox, reset, or stop the timer, or read a signal notification register).

Access Type	Read-blocking
Channel Number	x‘0’



Bits	Field Name	Description
0:18	Reserved	Reserved.
19	Ms	Multisource Synchronization Event (see page 154) This event is triggered when the Multisource Synchronization request has completed. The Multisource Synchronization request completes when all pending transfers before a write of the MFC Write Multisource Synchronization Request Channel (see page 132) have completed. This event is triggered immediately if no transfers are pending at the time of the MFC Write Multisource Synchronization Request Channel write. 0 Event has not occurred 1 Event has occurred and has not been acknowledged
20	A	Privileged Attention Event (see page 154) This event is triggered by setting the SPU attention-event request bit in the SPU Privileged Control Register. Access to this register should be limited to privileged software. 0 Event has not occurred 1 Event has occurred and has not been acknowledged
21	Lr	Lock Line Reservation Lost Event (see page 153) This event is triggered when a get lock-line and reserve (**getllar**) command is issued, and the reservation is reset due to the modification of data in the same lock line by an outside entity. It is not be set due to a reservation reset by a local action. 0 Event has not occurred 1 Event has occurred and has not been acknowledged This event is set when a snoop external to the MFC causes a lock-line reservation to be reset. The event must *not* be set if the reservation is lost due to a local action.
22	S1	SPU Signal Notification 1 Available Event (see page 152) This event is triggered when a processor or a device writes to the SPU Signal Notification 1 Register of the corresponding SPU. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
23	S2	SPU Signal Notification 2 Available Event (see page 151) This event is triggered when a processor or a device writes to the SPU Signal Notification 2 Register of the corresponding SPU. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.

Bits	Field Name	Description
24	Le	SPU Outbound Mailbox Available Event (see page 150) This event is triggered when the SPU Write Outbound Mailbox Channel count becomes greater than or equal to an implementation dependent mailbox queue-empty threshold. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
25	Me	SPU Outbound Interrupt Mailbox Available Event (see page 150) This event is triggered when the SPU Write Outbound Interrupt Mailbox Channel count becomes greater than or equal to an implementation dependent interrupt-mailbox queue-empty threshold. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
26	Tm	SPU Decrementer Event (see page 149) This event is triggered by the transition of the most significant bit of the SPU decrementer count from ‘0’ to ‘1’. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
27	Mb	SPU Inbound Mailbox Available Event (see page 148) This event is triggered when the SPU Read Inbound Mailbox Channel count becomes ‘0’. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
28	Qv	MFC SPU Command Queue Available Event (see page 147) This event is triggered by the transition of the MFC SPU command queue from a full to a not-full state. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
29	Reserved	Reserved
30	Sn	MFC DMA List Command Stall-and-Notify Event (see page 146) This event occurs when the MFC encounters one or more MFC list commands with the stall-and-notify flag set in the list elements (see *Section 7.4 DMA List Elements* beginning on page 53). When this type of MFC command is encountered, the list element is completed, and further list processing is suspended until the stall is acknowledged by the SPU program. 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.
31	Tg	MFC Tag-Group Status Update Event (see page 146) The tag status event occurs when MFC Read Tag-Group Status Channel (see page 117) is updated based on the tag status update requested by writing the MFC Write Tag Status Update Request Channel (see page 116). 0 Event has not occurred. 1 Event has occurred and has not been acknowledged.

Implementation Notes:

Hardware determines events by detecting the appropriate channel counts, decrementer count, or SPU Channel Access operation:

- The MFC Tag-Group Status Update Event is set when the count for the MFC Read Tag-Group Status Channel (see page 117) changes from ‘0’ to a non-zero value.
- The MFC DMA List Command Stall-and-Notify Event is set when the count for the MFC Read List Stall-and-Notify Tag Status Channel (see page 118) changes from ‘0’ to a non-zero value.
- The MFC SPU Command Queue Available Event is set when the count for the queued MFC Command Opcode Register changes from ‘0’ (full) to a non-zero (not full).
- The SPU Inbound Mailbox Available Event is set when the count for the SPU Read Inbound Mailbox Channel (see page 124) changes from ‘0’ to a non-zero value.

- The SPU Decrementer Event is set when the most significant bit of the decrementer count changes from ‘0’ to ‘1’. If a value loaded into the decrementer causes a change from ‘0’ to ‘1’ in the MSb, an event is signaled immediately. Setting the decrementer to a value of ‘0’ results in an event after a single decrementer interval.
- The SPU Outbound Mailbox Available Event is set when the SPU Write Outbound Interrupt Mailbox Channel count changes from a ‘0’ to a nonzero value.
- The SPU Outbound Interrupt Mailbox Available Event is set when the SPU Write Outbound Interrupt Mailbox Channel count changes from ‘0’ to a nonzero value.
- The SPU Signal Notification 2 Available Event is set when the count for the SPU Signal Notification 2 Channel (see page 127) changes from ‘0’ to a nonzero value.
- The SPU Signal Notification 1 Available Event is set when the count for the SPU Signal Notification 1 Channel (see page 126) changes from ‘0’ to a nonzero value.
- The Privileged Attention Event is set when the SPU Privileged Control Register (see page 215) is written with the attention-event request bit set to ‘1’.
- The Multisource Synchronization Event is set when MFC Write Multisource Synchronization Request Channel (see page 132) count changes from a value of ‘0’ to ‘1’.

9.11.2 SPU Write Event Mask Channel (SPU_WrEventMask)

The SPU Write Event Mask Channel selects which pending events affect the state of the SPU Read Event Status Channel (see page 136). The contents of this channel are retained until a subsequent channel write or SPU Channel access occurs. The current contents of this channel can be accessed by reading the SPU Read Event Mask Channel (see page 142).

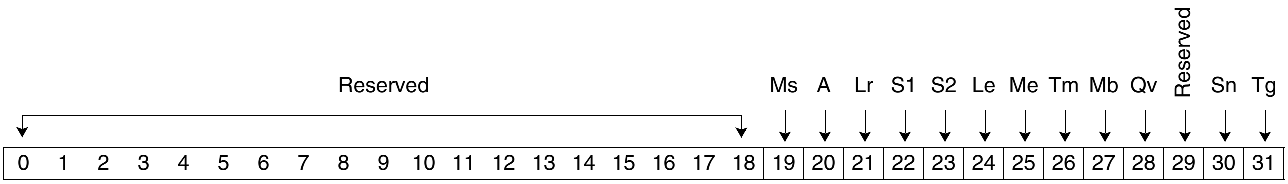
All events are recorded in the SPU Pending Event Register, regardless of the SPU event mask setting. Events remain pending until cleared by a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel (see page 144) or privileged software loads the SPU Pending Event Register with a new value using the SPU Channel Access Facility (see page 218). A pending event is cleared even if it is disabled.

Pending events, which are disabled and subsequently cleared, are not reflected in the SPU Read Event Status Channel (see page 136). Enabling a pending event results in an update of the SPU Read Event Status Channel and an SPU interrupt, if enabled.

This channel is non-blocking and does not have an associated count. A read channel count (**rchcnt**) instruction of this channel always returns ‘1’.

Access Type Write

Channel Number x‘1’



Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
0:18	Reserved	Reserved.
19	Ms	Multisource Synchronization Event enable. 0 Event is disabled. 1 Event is enabled.
20	A	Privileged Attention Event enable. 0 Event is disabled. 1 Event is enabled.
21	Lr	Lock Line Reservation Lost Event enable. 0 Event is disabled. 1 Event is enabled.
22	S1	SPU Signal Notification 1 Available Event enable. 0 Event is disabled. 1 Event is enabled.
23	S2	SPU Signal Notification 2 Available Event enable. 0 Event is disabled. 1 Event is enabled.

Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
24	Le	SPU Outbound Mailbox Available Event enable. 0 Event is disabled. 1 Event is enabled.
25	Me	SPU Outbound Interrupt Mailbox Available Event enable. 0 Event is disabled. 1 Event is enabled.
26	Tm	SPU Decrementer Event enable. Setting this bit to ‘0’ before acknowledging a decrementer event results in the decrementer being stopped, regardless of the decrementer event status. See *Section 9.7* on page 128 for more details 0 Event is disabled. 1 Event is enabled.
27	Mb	SPU Inbound Mailbox Available Event enable, written by the PPE. 0 Event is disabled. 1 Event is enabled.
28	Qv	MFC SPU Command Queue Available Event enable. 0 Event is disabled. 1 Event is enabled.
29	Reserved	Reserved.
30	Sn	MFC DMA List Command Stall-and-Notify Event enable. 0 Event is disabled. 1 Event is enabled.
31	Tag	MFC Tag-Group Status Update Event enable. 0 Event is disabled. 1 Event is enabled.

Implementation Note:

The SPU Decrementer (see page 128) must stop if the SPU Decrementer Event (bit 26) is disabled when an SPU Decrementer Event is acknowledged. The SPU Decrementer Event is acknowledged by writing a ‘1’ to bit 26 of the SPU Write Event Acknowledgment Channel (see page 144). The decrementer must stop regardless of the status of the SPU Decrementer Event at the time of the acknowledgement. The SPU Decrementer must not stop if the SPU Decrementer Event is enabled when the event is acknowledged. Once the SPU Decrementer is stopped, no further SPU Decrementer Events can occur.

9.11.3 SPU Read Event Mask Channel (SPU_RdEventMask)

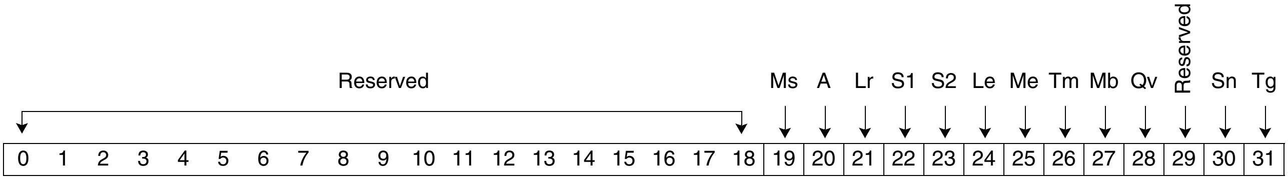
The SPU Read Event Mask Channel is used to read the current value of the Event Status Mask. Reading this channel always returns the last data written by the SPU Write Event Mask Channel (see page 140).

The SPU Read Event Mask Channel allows software to read the state of the Event Status Mask. This channel can be used to avoid software shadow copies of the Event Status Mask and for an SPE context save and restore operations.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type Read

Channel Number x‘B’



Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
0:18	Reserved	Reserved.
19	Ms	Multisource Synchronization Event enable. 0 Event is disabled. 1 Event is enabled.
20	A	Privileged Attention Event enable. 0 Event is disabled. 1 Event is enabled.
21	Lr	Lock Line Reservation Lost Event enable. 0 Event is disabled. 1 Event is enabled.
22	S1	SPU Signal Notification 1 Available Event enable. 0 Event is disabled. 1 Event is enabled.
23	S2	SPU Signal Notification 2 Available Event enable. 0 Event is disabled. 1 Event is enabled.
24	Le	SPU Outbound Mailbox Available Event enable. 0 Event is disabled. 1 Event is enabled.
25	Me	SPU Outbound Interrupt Mailbox Available Event enable. 0 Event is disabled. 1 Event is enabled.
26	Tm	SPU Decrementer Event enable. 0 Event is disabled. 1 Event is enabled.

Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
27	Mb	SPU Inbound Mailbox Available Event enable, written by the PPE. 0 Event is disabled. 1 Event is enabled.
28	Qv	MFC SPU Command Queue Available Event enable. 0 Event is disabled. 1 Event is enabled.
29	Reserved	Reserved.
30	Sn	MFC DMA List Command Stall-and-Notify Event enable. 0 Event is disabled. 1 Event is enabled.
31	Tag	MFC Tag-Group Status Update Event enable. 0 Event is disabled. 1 Event is enabled.

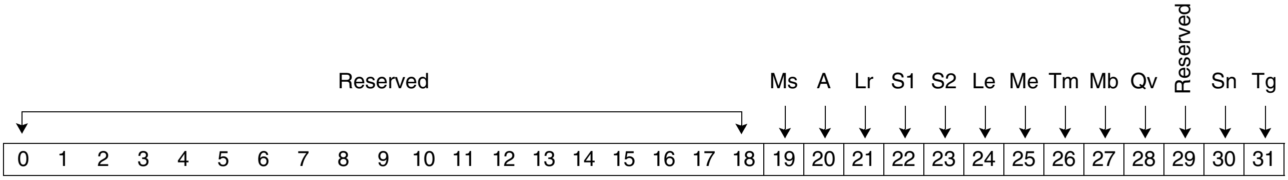
9.11.4 SPU Write Event Acknowledgment Channel (SPU_WrEventAck)

A write to the SPU Write Event Acknowledgment Channel, with specific event bits set, acknowledges that the corresponding events is being serviced by the software. Events that have been acknowledged are reset and resampled. Events that have been reported, but not acknowledged continue to be reported until acknowledged, or until cleared by privileged software using the SPU Channel Access Facility (see page 218).

Disabled events are not reported in the SPU Read Event Status Channel (see page 136) but they are held pending until they are cleared by writing a '1' to the corresponding bit in the SPU Write Event Acknowledgment Channel. Acknowledging a disabled event clears the event, even though it has not been reported. Clearing an event before it occurs, results in a software induced deadlock. Software should be careful in clearing unreported events. For more on handling these events, see *Section 9.11 SPU Event Facility* beginning on page 133 and for more on the events themselves, see *Section 9.12 SPU Event Definitions* beginning on page 145.

This channel is non-blocking and does not have an associated count. Whenever a read channel count (**rchcnt**) instruction is sent to this channel, the count is always returned as ‘1’.

Access Type	Write
Channel Number	x‘2’



Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
0:18	Reserved	Reserved.
19	Ms	Multisource Synchronization Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
20	A	Privileged Attention Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
21	Lr	Lock Line Reservation Lost Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
22	S1	SPU Signal Notification 1 Available Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
23	S2	SPU Signal Notification 2 Available Event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
24	Le	SPU Outbound Mailbox Available Event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.

Bits	Field Name	**Description.** For more information on these events, see *Section 9.12 SPU Event Definitions* beginning on page 145.
25	Me	SPU Outbound Interrupt Mailbox Available Event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
26	Tm	SPU Decrementer Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
27	Mb	SPU Inbound Mailbox Available Event acknowledgment 0 Event not acknowledged. 1 Event acknowledged.
28	Qv	MFC SPU Command Queue Available Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
29	Reserved	Reserved.
30	Sn	MFC DMA List Command Stall-and-Notify Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.
31	Tg	MFC Tag-Group Status Update Event acknowledgment. 0 Event not acknowledged. 1 Event acknowledged.

Implementation Note:

The SPU Decrementer stops if the SPU Decrementer Event (bit 26) is disabled when an SPU Decrementer Event is acknowledged. The SPU Decrementer Event is acknowledged by writing a ‘1’ to bit 26 of the SPU Write Event Acknowledgment Channel. The decrementer stops regardless of the status of the SPU Decrementer Event at the time of the acknowledgment. The decrementer does not stop if the SPU Decrementer Event is enabled when the event is acknowledged. Once the decrementer is stopped, no further SPU Decrementer Events occur. See *Section 9.7.1* beginning on page 128 for more details.

9.12 SPU Event Definitions

These events are:

- MFC Tag-Group Status Update Event (see page 146)
- MFC DMA List Command Stall-and-Notify Event (see page 146)
- MFC SPU Command Queue Available Event (see page 147)
- SPU Inbound Mailbox Available Event (see page 148)
- SPU Decrementer Event (see page 149)
- SPU Outbound Interrupt Mailbox Available Event (see page 150)
- SPU Outbound Mailbox Available Event (see page 150)
- SPU Signal Notification 2 Available Event (see page 151)
- SPU Signal Notification 1 Available Event (see page 152)
- Lock Line Reservation Lost Event (see page 153)
- Privileged Attention Event (see page 154)
- Multisource Synchronization Event (see page 154)

9.12.1 MFC Tag-Group Status Update Event

The MFC Tag-Group Status Update Event is used to notify an SPU program that a tag group or groups have completed, and that the MFC Read Tag-Group Status Channel (see page 117) has been updated and can be read without stalling the SPU. See *Section 9.3 MFC Tag-Group Status Channels* beginning on page 111 for more information.

The event occurs when the channel count for the MFC Read Tag-Group Status Channel changes from ‘0’ to ‘1’. When this event occurs, it sets Pend_Event[Tg] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Tg] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Tg] is set to ‘1’.

The Pend_Event[Tg] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel, or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

This event must be cleared before issuing any commands for the tag group or groups. For more information on the procedure for using the SPU Tag-Group Status Update event, see *Section 9.3 MFC Tag-Group Status Channels* beginning on page 111.

9.12.2 MFC DMA List Command Stall-and-Notify Event

The MFC DMA List Command Stall-and-Notify Event is used to notify an SPU program that a list element within an MFC DMA list command has completed and that the MFC Read List Stall-and-Notify Tag Status Channel (see page 118) has been updated and can be read without stalling the SPU. See *Section 7.4 DMA List Elements* beginning on page 53 for more information.

The event occurs when the channel count for the MFC Read List Stall-and-Notify Tag Status Channel changes from ‘0’ to ‘1’. The count is set to ‘1’ when all the transfers of the list elements with the stall-and-notify flag set, as well as the transfers for all the previous list elements in the MFC DMA list command, are complete with respect to the associated SPE. When this event occurs, it sets Pend_Event[Sn] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Sn] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Sn] is set to ‘1’.

The Pend_Event[Sn] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Sn bit set (that is, SPU_WrEventAck[Sn] is set to ‘1’) or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the MFC DMA List Command Stall-and-Notify Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
2. Mask the event by issuing a write channel instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Sn] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Sn] set to ‘1’.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

4. Perform a read channel (**rdch**) instruction to the MFC Read List Stall-and-Notify Tag Status Channel MFC_RdListStallStat[gn].
5. Use this information to determine which tag group or tag groups have a DMA List Element in the Stall and Notify state.
6. Perform the application-specific action with respect to each tag group having a stalled DMA list element.

   Note: When a DMA list contains multiple list elements having the Stall and Notify flag set, or when a Tag Group has multiple DMA list commands queued with elements having the Stall and Notify flag set, it is essential for the application software to initialize to 0 a tag-group specific stall counter before the DMA list commands are queued for the tag group. In addition, when multiple DMA list commands are queued for a tag group with Stall and Notify elements, ordering must be enforced with tag-specific fences, barriers, or the command barrier. Each time a Stall and Notify status is indicated for a tag group, the corresponding counter should be incremented. Application software can then use this counter to determine at what point in the list the stall has occurred. Application software uses stall and notify to update list element addresses and transfer sizes that follow the list element that has stalled due to dynamically changing conditions. List elements after the stalled list element can be skipped by setting their transfer sizes to 0. However the number of list elements in a queued DMA list command cannot be changed.
7. Acknowledge and resume each stalled DMA list command by issuing a write channel (**wrch**) instruction to the MFC Write List Stall-and-Notify Tag Acknowledgment Channel (MFC_WrListStallAck[MFC Tag]) where the supplied MFC Tag is the encoded Tag ID of the tag group to be resumed.
8. Exit the DMA List Stall and Notify handler.

   Note: If application software does not acknowledge all stalled tag groups indicated in the MFC_RdListStallStat[gn] channel, a second stall and notify event does not occur for the unacknowledged tag group.
9. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Sn] set to ‘1’.[1]
10. Exit the general event handler.[1]

9.12.3 MFC SPU Command Queue Available Event

The MFC SPU Command Queue Available Event is used to notify an SPU program that an entry in the MFC SPU Command Queue is available and that the MFC Command Opcode Channel (see page 103) can be written without stalling the SPU.

The event occurs when the channel count for the MFC Command Opcode Channel changes from a ‘0’ (full) to a nonzero (not full) value. The count is set to ‘1’ when one or more MFC DMA commands in the MFC SPU command queue is completed. When this event occurs, it sets Pend_Event[Qv] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Qv] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Qv] is set to ‘1’.

The Pend_Event[Qv] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Qv bit set (that is, SPU_WrEventAck[Qv] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

The procedure for handling the MFC SPU Command Queue Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the "mask".[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Qv] set to '0'.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Qv] set to '1'.
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the MFC Command Opcode Channel (MFC_Cmd).
5. If the channel count is '0', skip to step 8.
6. Enqueue a DMA command to the MFC command queue.
7. If more commands are left to queue, return to step 3.
8. Exit the SPU Command Queue handler.
9. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Qv] set to '1'.[1]
10. Exit the general event handler.[1]

9.12.4 SPU Inbound Mailbox Available Event

The SPU Inbound Mailbox Available Event is used to notify an SPU program that a PPE or other device has written to an empty SPU mailbox and that the SPU Read Inbound Mailbox Channel (see page 124) can be read without stalling the SPU.

The event occurs when the channel count for the SPU Read Inbound Mailbox Channel changes from a '0' (empty) to a nonzero (not empty) value. When this event occurs, it sets Pend_Event[Mb] to '1'. If this event is enabled (that is, SPU_RdEventMask[Mb] is '1'), the count for the SPU Read Event Status Channel is set to '1' and SPU_RdEventStat[Mb] is set to '1'.

The Pend_Event[Mb] bit is set to '0' when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Mb bit set (that is, SPU_WrEventAck[Mb] is set to '1') or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to '0'.

The procedure for handling the SPU Inbound Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in "mask".[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask(Mb) set to '0'.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Mb] set to '1'.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

4. Obtain a channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Read Inbound Mailbox Channel.
5. If the channel count is ‘0’, skip to step 8.
6. Read next mailbox data entry by issuing a read channel (**rdch**) instruction to the SPU Read Inbound Mailbox Channel (SPU_RdInMbox).
7. Return to step 3.
8. Exit the SPU inbound mailbox handler.
9. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Mb] set to ‘1’.[1]
10. Exit the general event handler.[1]

9.12.5 SPU Decrementer Event

The SPU Decrementer Event is used to notify an SPU program that the decrementer has reached ‘0’. See *Section 9.7.1 SPU Write Decrementer Channel* beginning on page 128 for more information.

The event occurs when the most-significant bit of the decrementer changes from ‘0’ to ‘1’ (negative) value. When this event occurs, it sets Pend_Event[Tm] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Tm] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Tm] is set to ‘1’.

The Pend_Event[Tm] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Tm bit set (that is, SPU_WrEventAck[Tm] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the SPU Decrementer Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Tm] set to ‘0’.[1]
3. Acknowledge the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel (SPU_WrEventAck[Tm] set to ‘1’).
4. Read the decrementer value by issuing a read channel (**rdch**) instruction to the SPU Read Decrementer Channel. If this value is negative, it can be used to determine how much additional time has elapsed from the desired interval.
5. If a new timer event is desired, write (**wrch**) a new decrementer value to the SPU Write Decrementer Channel.
6. Exit the SPU decrementer event handler.
7. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Tm] set to ‘1’.[1]
8. Exit the general event handler.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

9.12.6 SPU Outbound Interrupt Mailbox Available Event

The SPU Outbound Interrupt Mailbox Available Event is used to notify an SPU program that a PPE or another device has read from a full SPU Outbound Interrupt Mailbox Register (see page 213) and that the SPU Write Outbound Interrupt Mailbox Channel (see page 123) can be written without stalling the SPU.

The event occurs when the channel count for the SPU Write Outbound Interrupt Mailbox Channel changes from a ‘0’ (full) to a non-zero (not full) value. When this event occurs, it sets Pend_Event[Me] to ‘1’. If this event is enabled (that is, SPU_RdEventMask[Me] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Me] is set to ‘1’.

The Pend_Event[Me] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Me bit set (that is, SPU_WrEventAck[Me] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the SPU Outbound Interrupt Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in the “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Me] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Me] set to a ‘1’.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Write Outbound Interrupt Mailbox Channel (see page 123).
5. If channel count is ‘0’, skip to step 7.
6. Write a new mailbox data entry by issuing a write channel (**wrch**) instruction to the SPU Write Outbound Interrupt Mailbox Channel (see page 123).
7. Exit the SPU Outbound Interrupt Mailbox Available handler.
8. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Me] set to ‘1’.[1]
9. Exit the general event handler.[1]

9.12.7 SPU Outbound Mailbox Available Event

The SPU Outbound Mailbox Available Event is used to notify an SPU program that either a processor or another device has read from a full SPU Outbound Mailbox Register (see page 91) and that the SPU Write Outbound Mailbox Channel (see page 122) can be written without stalling the SPU.

The event occurs when the channel count for the SPU Write Outbound Mailbox Channel (see page 122) changes from a ‘0’ (full) to a non-zero (not full) value. When this event occurs, it sets Pend_Event[Le] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Le] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Le] is set to ‘1’.

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

The Pend_Event[Le] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel (see page 144) with the Le bit set to ‘1’ (that is, SPU_WrEventAck[Le] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the SPU Outbound Mailbox Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Le] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Le] set to ‘1’.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Write Outbound Mailbox Channel.
5. If the channel count is ‘0’, skip to step 7.
6. Write a new mailbox data entry by issuing a write channel (**wrch**) instruction to the SPU Write Outbound Mailbox Channel.
7. Exit the SPU Outbound Mailbox handler.
8. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Le] set to ‘1’.[1]
9. Exit the general event handler.[1]

9.12.8 SPU Signal Notification 2 Available Event

The SPU Signal Notification 2 Available Event is used to notify an SPU program that another processor or device has written to an empty SPU Signal Notification 2 Register and that the SPU Signal Notification 2 Channel can be read without stalling the SPU. See *Section 9.6 SPU Signalling Channels* beginning on page 125 for more information.

The event occurs when the channel count for the SPU Signal Notification 2 Channel changes from ‘0’ (empty) to ‘1’ (valid) value. When this event occurs, it sets Pend_Event[S2] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[S2] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[S2] is set to ‘1’.

The Pend_Event[S2] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the S2 bit set (that is, SPU_WrEventAck[S2] is set to ‘1’) or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

The procedure for handling the SPU Signal Notification 2 Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in "mask".[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[S2] set to '0'.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[S2] set to a '1'.[1]
4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 2 Channel (see page 127).
5. If the channel count is '0', skip to step 7.
6. Read the signal data by issuing a read (**rdch**) channel instruction to the SPU Signal Notification 2 Channel.
7. Exit the Signal Notification 2 handler.
8. Restore the "mask" by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[S2] set to '1'.[1]
9. Exit the general event handler.[1]

9.12.9 SPU Signal Notification 1 Available Event

The SPU Signal Notification 1 Available Event is used to notify an SPU program that another processor or device has written to an empty SPU Signal Notification 1 Register and that the SPU Signal Notification 1 Channel (see page 126) can be read without stalling the SPU. See *Section 9.6 SPU Signalling Channels* beginning on page 125 for more information.

The event occurs when the channel count for the SPU Signal Notification 1 Channel changes from a '0' (empty) to a '1' (valid) value. When this event occurs, it sets Pend_Event[S1] to '1'. If the event is enabled (that is, SPU_RdEventMask[S1] is set to '1'), the count for the SPU Read Event Status Channel is set to '1' and SPU_RdEventStat[S1] is set to '1'.

The Pend_Event[S1] bit is set to '0' when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the S1 bit set (that is, SPU_WrEventAck[S1] is set to '1'), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to '0'.

The procedure for handling the SPU Signal Notification 1 Available Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in "mask".[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[S1] set to '0'.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[S1] set to '1'.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

4. Obtain the channel count by issuing a read channel count (**rchcnt**) instruction to the SPU Signal Notification 1 Channel (see page 126).
5. If the channel count is ‘0’, skip to step 7.
6. Read the signal data by issuing a read channel (**rdch**) instruction to the SPU Signal Notification 1 Channel (SPU_RdSigNotify1).
7. Exit the Signal Notification 1 handler.
8. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[S1] set to ‘1’.[1]
9. Exit the general event handler.[1]

9.12.10 Lock Line Reservation Lost Event

The Lock Line Reservation Lost Event is used to notify an SPU program of a bus action that has resulted in the loss of the reservation on a cache line. A reservation is acquired by an SPU program by issuing a **getllar** command. The reservation is lost when another processor or device modifies the cache line with the reservation. The reservation can also be lost if privileged software writes the flush bit in the MFC Atomic Flush register (MFC_Atomic_Flush[F] is set to ‘1’). See *Section 7.8.1 Get Lock Line and Reserve Command* beginning on page 60 for more information.

The event occurs when the reservation is lost. When this event occurs, it sets Pend_Event[Lr] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Lr] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Lr] is set to ‘1’.

The Pend_Event[Lr] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Lr bit set (that is, SPU_WrEventAck[Lr] is set to ‘1’) or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the Lock Line Reservation Lost Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[1]
2. Mask the event by issuing a write channel instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Lr] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Lr] set to ‘1’.[1]
4. Perform the application-specific function in response to system modification of data in the lock line area.

   This is usually started by checking a software structure in memory to determine if a lock line is still being monitored. If it is still being “waited on,” then the next step would typically consist of issuing a **getllar** command to the same lock line area that was modified to obtain the new data and then act on that data.
5. Exit the Lock Line Reservation Lost Event handler.
6. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Lr] set to ‘1’.[1]
7. Exit the general event handler.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

9.12.11 Privileged Attention Event

The Privileged Attention Event is used to notify an SPU program that privileged software is requesting attention from an SPU program. Privileged software requests attention by writing ‘1’ to the attention event required bit in the SPU Privileged Control Register (see page 215) (that is, SPU_PrivCntl[A] is set to ‘1’).

The event occurs when SPU_PrivCntl[A] is set to ‘1’. When this event occurs, it sets Pend_Event[A] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[A] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[A] is set to ‘1’.

The Pend_Event[A] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the A bit set (that is, SPU_WrEventAck[A] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The procedure for handling the Privileged Attention Event is:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[A] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[A] set to ‘1’.[1]
4. Perform the application-specific function in response to a Privileged Attention Event.

   This can be used to signal that a yield of the SPU is being requested or some other action. An application or operating system-specific response to the Privileged Attention Event should be issued, such as stop and signal, SPU Inbound mailbox write, SPU Outbound Interrupt mailbox write, or an update of a status in system or I/O memory space.
5. Exit the Privileged Attention Event handler.
6. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[A] set to ‘1’.[1]
7. Exit the general event handler.[1]

9.12.12 Multisource Synchronization Event

The Multisource Synchronization Event is used to notify an SPU program that a Multisource Synchronization request has completed. A multisource synchronization is requested by writing (**wrch**) to the MFC Write Multisource Synchronization Request Channel (see page 132).

The event occurs when the channel count for the MFC Write Multisource Synchronization Request Channel changes from ‘0’ to ‘1’. When this event occurs, it sets Pend_Event[Ms] to ‘1’. If the event is enabled (that is, SPU_RdEventMask[Ms] is set to ‘1’), the count for the SPU Read Event Status Channel is set to ‘1’ and SPU_RdEventStat[Ms] is set to ‘1’.

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

The Pend_Event[Ms] bit is set to ‘0’ when a channel write (**wrch**) is issued to the SPU Write Event Acknowledgment Channel with the Ms bit set (that is, SPU_WrEventAck[Ms] is set to ‘1’), or when privileged software updates the SPU Pending Event Register using the SPU Channel Access Facility with the corresponding bit set to ‘0’.

The Multisource Synchronization Event must be cleared before issuing the Multisource Synchronization request.

The procedure for handling the Multisource Synchronization Event is as follows:

1. Perform a read channel (**rdch**) instruction to the SPU Read Event Mask Channel and save the data in a “mask”.[1]
2. Mask the event by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Ms] set to ‘0’.[1]
3. Acknowledge the event by performing a write channel (**wrch**) instruction to the SPU Write Event Acknowledgment Channel with SPU_WrEventAck[Ms] set to ‘1’.[1]
4. Perform the application-specific function in response to the completion of a pending multisource synchronization operation.

   This would typically indicate that the data in a particular buffer has been completely updated, or that a buffer area is no longer in use.
5. Exit the Multisource Synchronization Event handler.
6. Restore the “mask” by issuing a write channel (**wrch**) instruction to the SPU Write Event Mask Channel with SPU_WrEventMask[Ms] set to ‘1’.[1]
7. Exit the general event handler.[1]

[1] When multiple events are enabled, a common handler should be used to save the current event mask, mask all events that are to be handled in one channel write, and acknowledge all events that are to be handled in a single channel write. Then each event specific handler should be invoked to handle the event. The common handler should then restore the current event mask from the saved value and exit. This technique minimizes the generation of spurious events.

10. Storage Access Ordering

As shown in *Figure 10-1* on page 158, there are multiple storage domains in the CBEA. The *PowerPC Architecture, Book II* defines storage access ordering and synchronization facilities for the main storage domain. The *Synergistic Processing Unit Instruction Set Architecture* defines storage access ordering and synchronization facilities for the local storage domain and channel domain.

In a CBEA-compliant processor, there can be multiple local storage and channel domains, but only one main storage domain. The ordering of accesses between these different domains is described in the following sections. This description assumes that the local storage is accessed from the main storage domain with the storage attribute of caching inhibited. The CBEA also defines DMA facilities in the Memory Flow Controller (MFC) for initiating storage accesses in both domains and additional synchronization facilities for main storage beyond those described in the *PowerPC Architecture, Book II*. Also see *Section 6 Memory Flow Controller* beginning on page 43, *Section 7 MFC Commands* beginning on page 47, *Section 8 Problem State Memory-Mapped Registers* beginning on page 69, and *Section 9 Synergistic Processor Unit Channels* beginning on page 99 for more details on these facilities.

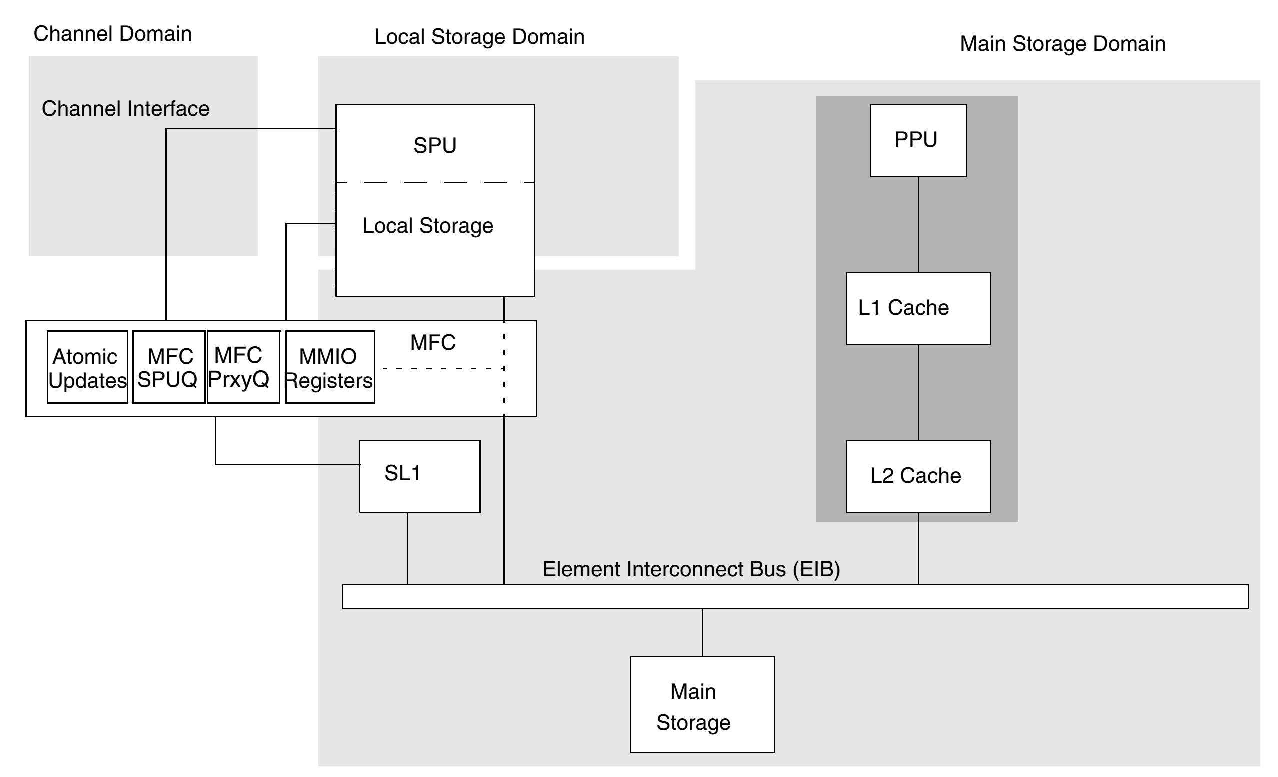
A Synergistic Processor Unit (SPU) initiates accesses within the local storage and channel domains and a Power Processor Element (PPE) initiates accesses within the main storage domain. The MFC can initiate accesses in both the main storage and local storage domains. The local storage can have an alias in the main storage. Therefore, it is shown both in the local storage domain and in the main storage domain. Access to local storage using the alias adheres to the ordering rules for the main storage domain.

Examples of the access ordering are given in *Appendix F Examples of Access Ordering* beginning on page 291.

As shown in *Figure 10-1* on page 158, the MFC maintains separate command queues for MFC SPU commands and MFC proxy commands.
MFC synchronization commands issued to the MFC proxy queue only arrange the order of the MFC commands within the MFC proxy command queue. MFC synchronization commands issued by the SPU only arrange the order of MFC commands within the MFC SPU command queue.

When an MFC synchronization command finishes processing, both the local storage access and the main storage access are performed. Later accesses initiated by the SPU either directly by loads, by stores to local storage, or indirectly by channel commands are therefore ordered.

Figure 10-1. Storage Domains in a CBEA-Compliant Processor



MFC: Memory Flow Controller
MFC SPUQ: Memory Flow Controller Synergistic Processor Unit Command Queue
MFC PrxyQ: Memory Flow Controller Proxy Command Queue
MMIO Registers: Memory Management Input-Output Registers
PPU: PowerPC Processor Unit
SL1: First-level cache for DMA transfers between local storage and main memory
SPU: Synergistic Processor Unit

The **putqlluc** MFC atomic command is placed into the MFC SPU command queue along with other commands. Since this command is queued, it executes independently of any pending immediate **getlar**, **putllc** and **putlluc** MFC atomic update commands.

The **putqlluc** command has an implied tag-specific **fence**; therefore, it uses the MFC tag parameter. To determine when the **putqlluc** command is complete, software must wait for the tag-group completion. When completed, all accesses from earlier issued commands with the same tag id in the MFC SPU command queue and the accesses for the **putqlluc** command are completed. Due to the implied tag-specific **fence**, the local storage and main storage accesses performed by the **putqlluc** are ordered with respect to all earlier issued commands with the same tag (tag-group) in MFC SPU command queue.

10.1 Order of Command Execution

Both *PowerPC Architecture, Book II* and *Synergistic Processing Unit Instruction Set Architecture* have a sequential execution model. In this model, from a software viewpoint, instructions appear to be executed in the order specified by the program, or program order. However, the order in which storage accesses are performed can be different from the program order.

The following terms are used in the description of storage access ordering:

Term	Definition
Main storage	Main memory, all local storages, and memory-mapped registers.
MFC EA access	Accesses to main storage using the effective address in the main storage domain.
MFC LSA access	Accesses to local storage using the local storage address in the local storage domain.
MFC put commands	The collection of all MFC commands that transfer data from local storage to the main storage domain. See *Section 7.6 Put Commands (Local Storage to Main Storage)* beginning on page 56.
MFC get commands	The collection of all MFC commands that transfer data from the main storage domain to local storage. See *Section 7.5 Get Commands (Main Storage to Local Storage)* beginning on page 54.

10.2 Main Storage Domain Access Ordering

The storage model for the CBEA is weakly consistent. This model incorporates the same weakly consistent model as the PowerPC Architecture supported by the PPE. This model provides an opportunity for improved performance over a model that has stronger consistency rules, but places the responsibility on the programmer or programming tools to ensure that ordering or synchronization instructions, commands, or command modifiers are properly placed when the storage is shared by multiple units in the CBEA. The PPE Storage Access Ordering and instructions are defined *PowerPC Architecture, Book II*. For DMA operations initiated by the Memory Flow Controllers found in the SPEs, storage ordering command modifiers, fence and barrier are provided as are the synchronization commands **mfcsync**, **mfceieio** and **barrier**. For more information on these facilities, see *Section 7.9 MFC Synchronization Commands* beginning on page 62.

The following rules apply to main storage access order in the main storage domain.

- If two PPU store instructions specify storage locations that are both caching inhibited and guarded, the corresponding storage accesses are performed in program order with respect to any PPE or other SPE's main storage accesses.
- All accesses resulting from a single MFC **put** command specifying storage that is both caching inhibited and guarded are performed in sequential (increasing) address order with respect to any PPE or other SPE's main storage accesses.
- All accesses resulting from a single MFC **put** list command specifying storage that is both caching inhibited and guarded are performed in sequential (increasing) address order for each list element and list order between elements with respect to any PPE or other SPE's main storage accesses.
- If a PPU load instruction depends on the value returned by a preceding load instruction (because the value is used to compute the effective address specified by the second load) the corresponding storage accesses are performed in program order with respect to any PPE or SPE to the extent required by the associated memory coherence required attributes.[1] This applies even if the dependency has no effect on program logic (for example, the value returned by the first load is ANDed with zero and then added to the effective address specified by the second load).

[1] The phrase "to the extent required by the associated memory coherence required attributes" refers to the memory coherence required attribute, if any, associated with each main storage access. This phrase does not apply to storage accesses in the local storage domain.

- If an MFC atomic command is followed by a read channel (**rdch**) from the MFC Read Atomic Command Status Channel that returns a status indicating completion of the command, the MFC atomic update access to main storage is performed before any main storage accesses specified by a MFC command, which was issued later by the SPU through the channel interface.
- When an MFC command queue process is suspended by an MMIO operation by a PPE, all earlier main storage accesses are performed with respect to the PPE causing the suspension before any later MFC EA accesses are performed when the MFC command queue process is no longer suspended. The MFC Multisource Synchronization Facility (see page 96) defines a facility that can be used when it is necessary to ensure that all earlier MFC EA accesses are performed with respect to any and all PPU and MFC EA accesses.
- When the PPU executes a **sync** or **eieio** instruction, a memory barrier is created, which orders applicable storage accesses. When an MFC performs an **mfcsync** or **mfceieio** command, a memory barrier is created, which orders applicable EA storage accesses in pairs as described below.

  For example, let A be a set of storage accesses that includes all storage accesses associated with instructions or applicable MFC commands preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions or applicable MFC commands following the barrier-creating instruction. For each applicable pair a_i,b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any PPE or SPE, to the extent required by the associated Memory Coherence Required attributes,[1] before b_j is performed with respect to that PPE or SPE.

  The memory barrier orders main storage accesses performed by one unit. Some memory barriers have an additional cumulative property, which orders certain main storage accesses done by other units.

  For example, let P1 represent the initiator of main storage accesses and a memory barrier. The ordering done by the memory barrier is said to be “cumulative”, if it also orders main storage accesses that are performed by a PPE, or an SPE other than P1, as follows:

  - A includes all applicable main storage accesses by any PPE or SPE that have been performed with respect to P1 before the memory barrier is created.
  - B includes all applicable main storage accesses by any PPE or SPE that are performed after a load instruction executed by P1 processor or device has returned the value stored by a store that is in B.

  For storage accesses to storage that does not have the Cache Inhibited and Guarded attribute, no ordering should be assumed among the storage accesses caused by a single instruction (that is, by an instruction for which the access is not atomic), a single MFC command (that is, by an MFC command for which the access is not atomic), and no means are provided for controlling the order.

10.3 Local Storage Access Ordering

The following rules apply to local storage access order in the local storage domain. The access order for SPU channel reads and writes is also described. The MFC DMA commands perform a local storage access using the local storage address (LSA) parameter, which is referred to as an MFC LSA access.

- When an SPU executes either a **sync** or a **dsync** instruction, a memory barrier is created that arranges the SPU instructions in the following order:
  - Earlier load instructions
  - Earlier store instructions
  - Earlier channel read instructions that are performed before later-issued channel read instructions
  - Channel write instructions

  - Later load instructions
  - Later store instructions

- The MFC **mfcsync** and **mfceieio** commands that arrange the order of all earlier commands in the same queue and with the same tag relative to all later commands in the same command queue with the same tag because they have an implied tag specific barrier. MFC LSA accesses for all earlier commands are performed before MFC LSA accesses for any later commands, that is, those following the **mfcsync** and **mfceieio** commands.
- An MFC **barrier** command arranges the order of all earlier commands in the same queue relative to all later-issued commands in the same queue. MFC LSA accesses for all earlier commands are performed before MFC LSA accesses for any later commands, that is, those following the MFC **barrier** command.
- An MFC **put** or **get** command with a tag-specific **fence** arranges the order of all earlier commands with the same tag and in the same queue relative to this command. MFC LSA accesses for all earlier commands in the same queue and with the same tag are performed before any MFC LSA accesses for a MFC **put** or **get** command with a tag-specific **fence**.
- An MFC **put** or **get** command with a tag-specific **barrier** orders all earlier-issued commands in the same queue relative to all later-issued commands with the same tag and in the same queue. MFC LSA accesses for all earlier-issued commands in the same queue and with the same tag are performed before any MFC LSA accesses for later-issued commands with the same tag, including the command with the tag-specific **barrier**.
- The MFC LSA access is complete for a queued **put** or **get** command when a channel read instruction issued to the MFC Read Tag-Group Status Channel (see page 117) returns a status that indicates the tag group associated with the put or get command is complete.
- An immediate MFC atomic command that is followed by a read from the MFC Read Atomic Command Status Channel (see page 120) returns the status indicating a completion of the command. In this case, the MFC atomic update access involving local storage is completed before any later-issued SPU load or store instruction.
- All earlier MFC LSA accesses are performed when the MFC command queue process is suspended by a PPE MMIO operation before any later MFC LSA accesses performed once the MFC command queue process is resumed.

10.4 Cross-Domain Storage Access Order

Cross-domain storage access order specifies that the results of a strongly ordered access sequence in the main storage domain targeting a single SPE will be seen in the same relative order when accessed with a strongly ordered sequence in the local storage domain of that SPE. It also specifies that the results of a strongly ordered access sequence in the local storage domain of a specific SPE will be seen in the same relative order in the main storage domain access directed at that SPE. The SPU Inbound Mailbox Register (see page 92), the SPU Signal Notification 1 Register (see page 94) and SPU Signal Notification 2 Register (see page 95) are the only MMIO registers for which cross-domain storage access order applies.

Cross-domain storage access order is critical when a main storage update of SPE local storage is performed and the SPE is informed using an SPE Communication MMIO register.

For examples on how to use the cross-domain storage access order, see examples 15 through 18 in *Appendix F* .

10.5 Cumulative Access Ordering

The MFC Multisource Synchronization Register (see page 96) defines a facility to achieve cumulative ordering across the local storage and main storage domains.

10.6 MFC Overlapped Accesses

An MFC access to an effective address range that maps to its own local storage (the local storage alias) can produce valid results, even though the translated effective address area overlaps the local storage area. (For more information, see *Section 3.2.1.2 Local Storage Access Exceptions* on page 34.)

In the cases where valid results occur, the access specified by the local storage address (LSA) is still considered to be performed in the local storage domain and the local storage address specified by the effective address is still considered to be performed in the main storage domain. Ordering rules for each domain still apply.

10.7 Atomic Accesses

An access is single-copy atomic, or simply atomic, if it is always performed in its entirety with no visible fragmentation. Atomic accesses are therefore serialized; each happens in its entirety in some order, even when that order is not specified in the program, or enforced between processors.

The *Synergistic Processor Unit Instruction Set Architecture* defines atomicity in the SPU domain. All SPU loads and stores to local storage are atomic.

Single-Copy Atomicity (see page 34) describes atomicity in the main storage domain. All MFC and PPU accesses to local storage, which are defined as atomic in Single-Copy Atomicity, are also defined as atomic in local storage. Therefore, SPU, MFC, and PPU atomic accesses to local storage are performed in their entirety and no fragmentation of these accesses is observed by any other SPU, MFC, or PPU.

10.8 Store Combining

If two PowerPC store instructions specify storage locations that are both caching inhibited and not guarded, such stores can be combined into one access by a CBEA-compliant processor. If the two store instructions each were single-copy atomic and were combined by a CBEA-compliant processor, the combined store is not required to be single copy atomic. However the original single-copy atomicity of the original stores must be preserved. If the store instructions are separated by a **sync** or **eieio** instruction, combining cannot occur.

11. SPU Isolation Facility

The Cell Broadband Engine Architecture (CBEA) includes an optional facility that enables privileged software and applications to isolate and load a code image into one or more of the Synergistic Processor Units (SPUs). The SPU isolation facility ensures the code image loaded into the associated local storage has not been altered by any means.

This section describes only those aspects of the isolation facility that relate to the CBEA. Many details are implementation dependent. For more specific implementation information, contact the system or processor manufacturer.

The SPU isolation facility consists of bits within the registers and channels listed in this section and includes:

- SPU Run Control Register (see page 86)
- SPU Status Register (see page 87)
- SPU Read Machine Status Channel (see page 130)
- SPU Privileged Control Register (see page 215)

11.1 SPU Isolation Facility Features

The SPU Isolation Facility consists of six principal architectural features:

- An SPU isolation execution environment, independent for each SPU in the processor (architecture)
- An EXIT and LOAD function for moving between the SPU nonisolated execution environment and the SPU isolated execution environment
- An authentication and decryption master key
- A validation procedure for ensuring that the code image loaded when entering the SPU isolation execution environment has not been altered by any means
- Support for random number generation
- A persistent storage that retains its state between isolation sessions

The SPU isolation execution environment allows an application loaded into the isolated area of local storage to execute without being modified, observed, or compromised by any ordinary means external to that SPU. (The isolated area is discussed below.) In this environment, the SPU Next Program Counter does not control where the SPU starts executing instructions. The initial instruction executed when entering the SPU isolation execution environment is implementation-dependent.

When initiating a transition into an SPU isolation execution environment and while operating in this environment, an area of local storage starting at address zero is isolated from the system (isolated local storage area or isolated area). Only the associated SPU can access the isolated area. Access from all other processors and devices in the system must not be allowed. The size of the isolated area of local storage is implementation-dependent. The remaining area, or open area, of local storage is not isolated and remains accessible. The open area can be used for data transfers, control, and communications between the rest of the system and the SPU operating in the SPU isolation execution environment. In addition, internal and external accesses to all debug, test, performance monitoring, and diagnostic interfaces for the associated SPU must be disabled.

The SPU isolation facility provides two transition functions, EXIT and LOAD, to change the SPU between the two code execution environments: nonisolated and isolated. Initiating an EXIT function is required to change an SPU from an SPU isolation execution environment to an SPU nonisolated execution environment. The EXIT function erases all SPU states before exiting from the SPU isolation execution environment. Initiating a LOAD function is required to change an SPU from an SPU nonisolated execution environment to an SPU isolated execution environment. The LOAD function loads a code image into the local storage and begins executing the image in the SPU isolation execution environment. A CBEA-compliant processor can implement just the EXIT function, or both the EXIT and LOAD functions. The LOAD function cannot be implemented without an EXIT function.

The SPU isolation facility requires a CBEA-compliant processor with a nonvolatile storage for holding an authentication and decryption master key. The size of the nonvolatile storage is implementation-dependent. The non-volatile storage must not be accessible by any software, processor, or device in the system. The authentication and decryption master key is intended for exclusive use of the validation procedure.

As part of the LOAD function, a validation procedure is performed when entering the SPU isolation execution environment. This procedure must ensure no changes have been made to the code image by altering the external source image used to supply the code, by interfering with the loading operation, or by any other means. The validation procedure should use the authentication and decryption master key as part of the validation process. The validation procedure is implementation-dependent. For more information, contact the system or processor manufacturer.

The SPU isolation facility also requires support for a random number generation (RNG) function. The details of the RNG function is implementation-dependent. An implementation can either allow access to the RNG function in any operating environment or restrict access to it from any environment.

Since the EXIT function erases all information from the previous application, the SPU isolation facility requires a persistent storage that retains its value even after exiting the SPU isolation execution environment. The persistent storage is not required to be nonvolatile. The persistent storage enables an application to transfer data from one isolated session to another. Access to the persistent storage must only be allowed by an SPU operating in the SPU isolated environment. After a successful LOAD function, read and write access should be allowed to the persistent storage. An implementation must provide a method for an SPU application to change the access restrictions of the persistent storage after entering the SPU isolation execution environment (for example, by removing the read and write accesses). Once the access restrictions are changed, a new LOAD function must be initiated to restore the read and write accesses. The size of the persistent storage area and the method used to access the persistent storage is implementation-dependent.

Note: The persistent storage is typically associated with a physical SPU. Therefore, software must provide a method either to ensure that the physical SPU is not changed between two isolation sessions for a given application, or to virtualize the persistent storage. Furthermore, all due caution should be taken by software before loading any code not validated with the authentication and decryption master key to prevent malicious tampering with the persistent storage.

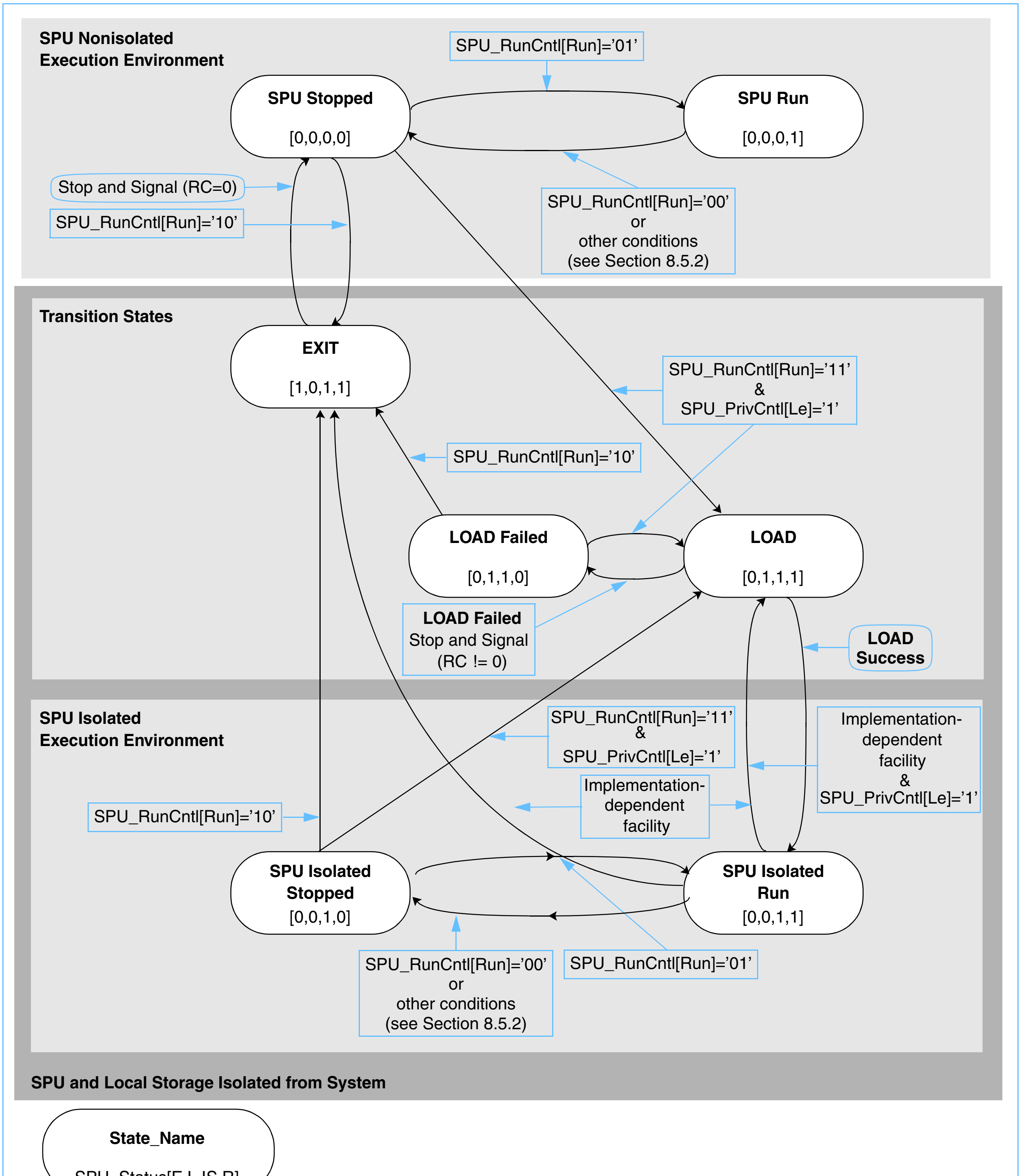
11.2 SPU Operating States

Support for the SPU isolation facility is optional for a CBEA-compliant processor. In addition, an implementation can implement a subset of the SPU isolation facility features, specifically the EXIT function. An SPU that implements the full isolation facility has seven states of operation compared to the two states for an implementation that does not support the isolation facility. If only the EXIT function is implemented, there are three operational states. All seven states with the optional SPU isolation facility implemented are:

- SPU stopped – Stopped in a nonisolated state (all SPU implementations)
- SPU run – Running in a nonisolated state (all SPU implementations)
- EXIT – Performing an isolated EXIT function (SPU with full or EXIT isolation support)
- LOAD – Performing an isolated LOAD function (SPU with full isolation support only)
- LOAD failed – The LOAD function failed (SPU with full isolation support only)
- SPU isolated run – SPU running in an isolated state (SPU with full isolation support only)
- SPU isolated stopped – SPU stopped in an isolated state (SPU with full isolation support only)

Figure 11-1 on page 166 is a state diagram illustrating the SPU states and the methods of transition between the states. The three lightly shaded boxes represent the two SPU operating environments (isolated and nonisolated) and the transition between these two environments. When the SPU is in any isolated or transition state, the SPU and the isolated area of local storage cannot be accessed from any other processing or data transfer resource within the system. This is shown as the darker shaded box around the figure. The diagram also includes the setting of several bits in the SPU Status Register used to identify the current SPU state. An application can control the transition between states by writing the SPU Run Control Register or an implementation-dependent facility accessible to an SPU operating in the SPU isolation execution environment. For simplicity, only the values written to the SPU Run Control Register that cause a state transitions are shown. All other values written to the SPU Run Control Register remain in their current state.

Figure 11-1. SPU State Transitions


SPU Nonisolated Execution Environment
SPU_RunCntl[Run]='01'
SPU Stopped
[0,0,0,0]
SPU Run
[0,0,0,1]
Stop and Signal (RC=0)
SPU_RunCntl[Run]='10'
SPU_RunCntl[Run]='00' or other conditions (see Section 8.5.2)
Transition States
EXIT
[1,0,1,1]
SPU_RunCntl[Run]='11' & SPU_PrivCntl[Le]='1'
SPU_RunCntl[Run]='10'
LOAD Failed
[0,1,1,0]
LOAD
[0,1,1,1]
LOAD Failed Stop and Signal (RC != 0)
LOAD Success
SPU Isolated Execution Environment
SPU_RunCntl[Run]='11' & SPU_PrivCntl[Le]='1'
Implementation-dependent facility & SPU_PrivCntl[Le]='1'
SPU_RunCntl[Run]='10'
Implementation-dependent facility
SPU Isolated Stopped
[0,0,1,0]
SPU Isolated Run
[0,0,1,1]
SPU_RunCntl[Run]='00' or other conditions (see Section 8.5.2)
SPU_RunCntl[Run]='01'
SPU and Local Storage Isolated from System
State_Name
SPU_Status[E,L,IS,R]

NOTE: The SPU Read Machine Status Isolation Status field (that is, SPU_RdMachStat[IS] is set to the same value as the SPU Status Isolation Status field (that is, SPU_Status[IS]).

Privileged Mode Environment

Section 12 through *Section 23* describe the instructions and facilities that are provided by the CBEA for operating environment software, such as an operating system or a hypervisor. The facilities in the rest of this document are only available to applications running in privileged mode, either Privilege 1 Mode or Privilege 2 Mode. Collectively, these facilities are referred to as the Privileged Mode Environment.

12. Overview

The organization of the Privileged Mode Environment is discussed next and each of the privileged facilities with their related registers are discussed in separate sections. *Table 12-1* on page 169, *Table 12-2* on page 172, and *Table 12-3* on page 174 list the privileged facilities.

- *Section 13 Compatibility with PowerPC Architecture, Book III* beginning on page 175
- *Section 14 Storage Addressing* beginning on page 177
- *Section 14.4 Real-Mode Storage Control Facilities (Optional)* beginning on page 193
- *Section 15 MFC Privileged Facilities* beginning on page 197
- *Section 16 SPU Privileged Facilities* beginning on page 215
- *Section 17 SPE Context Save and Restore* beginning on page 223
- *Section 18 PPE Address Range Facility* beginning on page 225
- *Section 19 Cache Replacement Management Facility* beginning on page 231
- *Section 20 Resource Allocation Management* beginning on page 235
- *Section 21 Interrupt Facilities* beginning on page 237
- *Section 22 Power Management* beginning on page 261
- *Section 23 Version Control* beginning on page 263.

12.1 Privileged Mode Facility Organization

The facilities described in the Privileged Mode Environment are classified as either Privilege 1 or Privilege 2. These designations relate to a suggested hierarchy of privileged access. The access hierarchy is defined to support a 2-level operating environment.

An example of such an operating environment is when multiple operating systems run concurrently on top of a more privileged hypervisor. This type of operating environment is called logical partitioning. Privilege 1 registers are the most privileged and are intended to be accessed by a hypervisor or by firmware operating in the
HV ='1' and PR = '0' mode (see the *PowerPC Architecture, Books I-III*), usually when supporting logical partitioning.

Privilege 2 facilities are intended for privileged operating system code running in the HV = '0' and PR = '0' mode. When a single level operating environment exists, firmware and the privileged operating system typically combine Privilege 1 and Privilege 2 resources into one privileged level. Access to facilities is not directly enforced by hardware. Privileged software should set the appropriate access privileges in the PowerPC address translation facility. Problem state software should never be allowed access to any facilities that are defined in the Privileged Mode Environment.

12.1.1 SPE Privilege 1 Facilities

Table 12-1 lists all CBEA-compliant SPE registers, which are designated Privilege 1 access. For information on each of the registers, see the relevant page.

Table 12-1. SPE Privilege 1 Memory Map (Page 1 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Control and Configuration Area			
x‘0000’	MFC_SR1	MFC State Register One (see page 197)	Read/Write
x‘0008’	MFC_LPID	MFC Logical Partition ID Register (see page 199)	Read/Write
x‘0010’	SPU_IDR	SPU Identification Register (see page 267)	Read/Write
x‘0018’	MFC_VR	MFC Version Register (see page 266)	Read Only
x‘0020’	SPU_VR	SPU Version Register (see page 265)	Read Only
x‘0128’:x‘00FF’	Reserved	Reserved	
Interrupt Area			
x‘0100’	INT_Mask_class0	Class 0 Interrupt Mask Register (see page 253)	Read/Write
x‘0108’	INT_Mask_class1	Class 1 Interrupt Mask Register (see page 254)	Read/Write
x‘0110’	INT_Mask_class2	Class 2 Interrupt Mask Register (see page 255)	Read/Write
x‘0118’:x‘013F’	Reserved	Reserved	Reserved
x‘0140’	INT_Stat_class0	Class 0 Interrupt Status Register (see page 257)	Read/Write
x‘0148’	INT_Stat_class1	Class 1 Interrupt Status Register (see page 258)	Read/Write
x‘0150’	INT_Stat_class2	Class 2 Interrupt Status Register (see page 259)	Read/Write
x‘0158’:x‘017F’	Reserved	Reserved	Reserved
x‘0180’	INT_Route	Interrupt Routing Register (see page 260)	Read/Write
x‘0188’:x‘01FF’	Reserved	Reserved	Reserved
Atomic Unit Control Area			
x‘0200’	MFC_Atomic_Flush	MFC Atomic Flush Register (see page 212) This is an implementation-dependent register	Read/Write
x‘0208’:x‘03FF’	SPU_Cache_ImplRegs	SPU cache hardware implementation-dependent registers. Refer to the specific implementation documentation.	

1. An implementation should support reading of these registers for diagnostic purposes.

Table 12-1. SPE Privilege 1 Memory Map (Page 2 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Translation Lookaside Buffer (TLB) Management Registers			
x‘0400’	MFC_SDR	MFC Storage Description Register (see page 200) Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x‘0408’:x‘04FF’	Reserved	Reserved	
x‘0500’	MFC_TLB_Index_Hint	TLB Index Hint Register (see page 186) Index of best TLB entry to update.	Read Only
x‘0508’	MFC_TLB_Index	TLB Index Register (see page 187) Index to TLB entry to update with TLB Real Page Number Register and TLB Virtual Page Number Register[1]	Write Only
x‘0510’	MFC_TLB_VPN	TLB Virtual Page Number Register (see page 188) Access to upper portion of TLB entry	Read/Write
x‘0518’	MFC_TLB_RPN	TLB Real Page Number Register (see page 189) Access to lower portion of TLB entry	Read/Write
x‘0520’:x‘053F’	Reserved	Reserved	
x‘0540’	MFC_TLB_Invalidate_Entry	TLB Invalidate Entry Register (see page 190) Virtual Page Number of TLB entry to invalidate[1] **Note:** Not available for PowerPC Processor Element (PPE).	Write Only
x‘0548’	MFC_TLB_Invalidate_All	TLB Invalidate All Register (see page 192) Invalidate all TLB entries (optional)[1] **Note:** Not available for PowerPC Processor Element (PPE).	Write Only
x'0550':'057F	Reserved	Reserved	
Memory Management (Implementation-dependent area: Refer to the specific implementation documentation.)			
x‘0580’:x‘05FF’	SPE_MMU_ImplRegs	SPE Memory Management Unit (MMU) Registers Refer to the specific implementation documentation for a description of this register.	
MFC Status and Control Area			
x‘0600’	MFC_ACCR	MFC Address Compare Control Register (see page 203)	Read/Write
x‘0610’	MFC_DSISR	MFC Data Storage Interrupt Status Register (see page 202)	Read/Write
x‘0620’	MFC_DAR	MFC Data Address Register (see page 201)	Read/Write
x‘0628’:x‘06FF’	Reserved	Reserved	
Replacement Management Table Area (RMT) (Implementation-dependent area. Refer to the specific implementation documentation)			
x‘0700’	MFC_TLB_RMT_Index	RMT Index Register (see page 233) Index of the replacement management tables.	Read/Write
x‘0710’	MFC_TLB_RMT_Data	RMT Data Register (see page 234). Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation-dependent.	Read/Write
x‘0718’:x‘07FF’	SPE_RMT_ImplRegs	SPE RMT hardware implementation-dependent registers (Same address listed with a different offset under memory management.)	

1. An implementation should support reading of these registers for diagnostic purposes.

Table 12-1. SPE Privilege 1 Memory Map (Page 3 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Command Data Storage Interrupt Area			
x‘0800’	MFC_DSIPR	MFC Data Storage Interrupt Pointer Register (see page 208) Contains a pointer to the command in the MFC command queue that caused the error condition.	Read Only
x‘0808’	MFC_LSACR	MFC Local Storage Address Compare Register (see page 205) 64-bit MFC Local Storage Address Compare Register	Read/Write
x‘0810’	MFC_LSCRR	MFC Local Storage Compare Result Register (see page 206) 64-bit MFC Local Storage Compare Results Register	Read Only
x‘0818’:x‘08FF’	Reserved	Reserved	
Real-Mode Support Registers			
x‘0900’	MFC_RMAB	MFC Real-Mode Address Boundary Register (see page 194)	Read/Write
x‘0908’:x‘0BFF’	Reserved	Reserved	
MFC Command Error Area			
x‘0C00’	MFC_CER	MFC Command Error Register (see page 207) Contains a pointer to the command in the DMA queue that caused the error condition.	Read Only
x‘0C08’:x‘0FFF’	Reserved	Reserved	
Implementation-Dependent Area (Refer to the specific implementation documentation for a detailed description of these registers)			
x‘1000’:x‘1FFF’	PV1_ImplRegs	Privilege 1 implementation-dependent registers	

[1] An implementation should support reading of these registers for diagnostic purposes.

12.1.2 SPE Privilege 2 Facilities

Table 12-2 contains a list of all CBEA-compliant registers, which are designated Privilege 2 access. For information on each of the registers, see the relevant page.

Table 12-2. SPE Privilege 2 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Registers			
x‘00000’:x‘010FF’	Reserved	Reserved	Reserved
Segment Lookaside Buffer Management Registers			
x‘01100’	Reserved	Reserved	Reserved
x‘01108’	SLB_Index	SLB Index Register (see page 179) Index of SLB entry to be updated by SLB_VSID and SLB_ESID ports[1]	Write Only
x‘01110’	SLB_ESID	SLB Effective Segment ID Register (see page 180) Access to the upper portion of an SLB entry	Read/Write
x‘01118’	SLB_VSID	SLB Virtual Segment ID Register (see page 181) Access to the lower portion of an SLB entry	Read/Write
x‘01120’	SLB_Invalidate_Entry	SLB Invalidate Entry Register (see page 182) ESID of SLB entry to invalidate[1]	Write Only
x‘01128’	SLB_Invalidate_All	SLB Invalidate All Register (see page 183) Invalidate all SLB entries[1]	Write Only
x‘01130’:x‘01FFF’	Reserved	Reserved	Reserved
(Context Save and Restore Area. Implementation-Dependent Area: Refer to the specific implementation documentation.**)**			
x‘02000’:x‘02FFF’	MFC_CSR_ImplRegs	MFC Context Save and Restore registers	
MFC Control			
x‘03000’	MFC_CNTL	MFC Control Register (see page 209)	Read/Write
x‘03008’:x‘03FFFF’	MFC_Cntl1_ImplRegs	Implementation-dependent control registers Refer to the specific implementation documentation.	
Interrupt Mailbox			
x‘04000’	SPU_OutIntrMbox	SPU Outbound Interrupt Mailbox Register (see page 213) SPU writes, PPE reads.	Read Only
SPU Control			
x‘04040’	SPU_PrivCntl	SPU Privileged Control Register (see page 215)	Read/Write
x‘04058’	SPU_LSLR	SPU Local Storage Limit Register (see page 217)	Read/Write
x‘04060’	SPU_ChnlIndex	SPU Channel Index Register (see page 218) This register selects which SPU channel in the specified SPU (n) is accessed using the SPU Channel Count Register or SPU Channel Data Register.	Read/Write
x‘04068’	SPU_ChnlCnt	SPU Channel Count Register (see page 220) This register is used to read or to initialize the SPU Channel Count Register selected by the SPU Channel Index Register.	Read/Write

[1] An implementation should support reading of these registers for diagnostic purposes

Table 12-2. SPE Privilege 2 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
x‘04070’	SPU_ChnlData	SPU Channel Data Register (see page 219) This register is used to read or to initialize the SPU channel data selected by the SPU Channel Index Register.	Read/Write
x‘04078’	SPU_Cfg	SPU Configuration Register (see page 221) This register is used to read or to set the configuration of the SPU Signal-Notification Registers in the specified SPU (n).	Read/Write
x‘04080’:x‘04FFF’	Reserved	Reserved	
(Implementation-Dependent Area. Refer to the specific implementation documentation for a detailed description of these registers)			
x‘05000’:x‘0FFFF’	PV2_ImplRegs	Privilege 2 implementation-dependent registers	
Reserved Area			
x‘10000’:x‘1FFFF’	Reserved	Reserved	

1. An implementation should support reading of these registers for diagnostic purposes

12.1.3 PPE Privilege 1 Facilities

Table 12-3 lists all CBEA-compliant PPE registers, which are designated Privilege 1 access. For information on each of the registers, see the relevant page.

Table 12-3. PPE Privilege 1 Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
Replacement-Management Table Area (RMT) (Implementation-Dependent Area: Refer to the specific implementation documentation.)			
x‘200’:x‘2FF’	Reserved	Reserved	
x‘300’	L2_RMT_Index	RMT Index Register (see page 233) Index of the replacement-management tables	Read/Write
x‘310’	L2_RMT_Data	RMT Data Register (see page 234) Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation-dependent.	Read/Write
x‘318’:x‘7FF’	Reserved	Reserved	
Implementation-Dependent Area (Refer to the specific implementation documentation for a detailed description of these registers.)			
x‘800’:x‘FFF’	PPEPV_ImplRegs	PPE Privilege 1 implementation-dependent registers	

13. Compatibility with PowerPC Architecture, Book III

This section covers the compatibility of facilities of the PPE as used in a CBEA-compliant system with a PPE as defined in the *PowerPC Architecture, Book III*. See the PowerPC Architecture document relating to each facility.

13.1 Optional Features in PowerPC Architecture, Book III (Required for CBEA)

The following facilities and instructions are considered optional in the *PowerPC Architecture, Book III*, but are required in the Cell Broadband Engine Architecture (CBEA).

- Real-mode storage control (see *Section 14.4 Real-Mode Storage Control Facilities (Optional)* on page 193 for more information).

13.2 Incompatibilities with PowerPC Architecture, Book III

Currently, there are no incompatibilities with *PowerPC Architecture, Book III*.

14. Storage Addressing

The storage addressing of the CBEA is compatible with the PowerPC Architecture. An address translation mechanism processes an effective address provided by an instruction or a memory flow controller (MFC) command. The mechanism for the PowerPC Processor Element (PPE) is described in *PowerPC Architecture, Book III.* The memory management unit (MMU) of the synergistic processing element (SPE) employs the same basic mechanism to process an effective address provided by an MFC command. If the instruction relocate (IR) and the data relocate (DR) bits in the PowerPC Machine State Register are set to '1', address translations are enabled for instructions and data fetches by the PPEs.

To enable translations of the effective addresses used in MFC commands, set the relocate (R) bit of the MFC State Register One (see page 197) to '1'.

Two steps are required to convert an effective address to a real address:

1. **Convert the effective address to a virtual address.** The conversion to a virtual address uses a segment lookaside buffer (SLB). The CBEA for the MMU differs from the PowerPC Architecture only in that the minimum number of SLB entries must be provided by an implementation. The PowerPC Architecture requires a minimum of 64 entries; the SPE MMU requires a minimum of eight SLB entries. The maximum number of SLB entries is 4 K. All implementations must have at least a minimum number of SLB entries and the associated management instructions, special purpose registers (SPRs), and memory-mapped I/O (MMIO) registers.
2. **Convert the virtual address to a real address**. The conversion of a virtual address to a real address uses a page table in main storage. The page table format and the conversion process are described in the *PowerPC Architecture, Book III.* In the software-managed mode, the TLB management instructions and registers must provide the capability to directly specify a virtual address to real address translation without the use of a hardware-accessed page table in main storage.

To enhance performance of these conversions, most implementations provide a Translation Lookaside Buffer Management (see page 184). The TLB is, basically, a special cache for keeping the recently used page table entries (PTEs). When operating in hardware-accessed page table mode, the TLB need not be kept consistent with the hardware-accessed page table in main storage. All implementations must support a virtual-to-real address translation mechanism using a hardware-accessed page table in main storage and a software-managed translation mechanism that uses the Translation Lookaside Buffer Management facilities (*Section 14.3 Translation Lookaside Buffer Management* beginning on page 184) to directly supply the translations without the need for a hardware-accessed page table in main storage.

Privileged software must manage the SLB in the CBEA for all PPE and SPE units utilizing address translation. The TLB is managed by hardware accesses to a page table in main storage in the PowerPC Architecture. The CBEA provides both a hardware-managed TLB mode and a privileged software-managed TLB mode. For both the PPE and the SPE, software management of the TLB allows the system software designer to forego the requirement of a hardware-accessible page table and use any format for the system page table; the format and size of the page table are not restricted by hardware, as required by the PowerPC Architecture. For more information on the hardware TLB management method, see *PowerPC Architecture, Book III*.

In addition, the software management facility can be used in combination with the hardware TLB management facility to preload translations into the TLB. For information on the TLB, see *Section 14.3 Translation Lookaside Buffer Management* beginning on page 184.

14.1 PPE Segment Lookaside Buffer Management

PPE SLBs are software managed. For management of the PPE SLBs, the PowerPC Architecture supports five instructions (**slbie**, **slbia**, **slbmte**, **slbmfev**, and **slbmfee**). Refer to *PowerPC Architecture, Book III* for a description of these instructions. For more information on the software managed SLBs for the SPE, see *Section 14.2.1 SLB Management* beginning on page 178.

14.2 SPE Segment Lookaside Buffer Management

The CBEA provides a set of MMIO registers for management of the SLBs in the synergistic processor element (SPE). These MMIO registers provide the same functionality for the SPE MMU as the PowerPC instructions have for the PPE MMU.

14.2.1 SLB Management

The SLB for each SPE is managed with a group of MMIO registers that mimic the source operands of the PowerPC SLB management instructions (that is, (**slbie**, **slbia**, **slbmte**, **slbmfev**, and **slbmfee**).

There is an SLB Index Register, an SLB Effective Segment ID Register, and an SLB Virtual Segment ID Register for each SPE. There is also an SLB Invalidate Entry Register and an SLB Invalidate All Register in each SPE used for invalidating SLB entries. For more information on these registers, see page 179 to page 183.

14.2.2 SLB Mapping

The SLB management registers for each SPE are mapped into the MMIO space of the main storage domain. The SLB management area must be accessed with storage attributes of caching inhibited and guarded and should be restricted to privileged code. Software must ensure that the following sequence of operations is performed individually and in the prescribed order.

Software must perform the following sequence to replace an SLB entry:

1. For each entry to be replaced:
   a. Set the index of the SLB entry to be replaced.
   b. Use the SLB Invalidate Entry Register (see page 182) to invalidate the SLB entry.
2. Set the new contents for the virtual segment ID (VSID) portion of the SLB entry.
3. Set the new contents for the effective segment ID (ESID) portion of the SLB entry along with the valid bit.

The contents of an SLB entry in the SPE MMU is accessed by using the SLB Effective Segment ID Register (see page 180) and the SLB Virtual Segment ID Register (see page 181). The SLB Index Register (see page 179) points to the SLB entry to be accessed by the SLB_ESID and SLB_VSID registers. The size and format of the SLB is implementation-dependent.

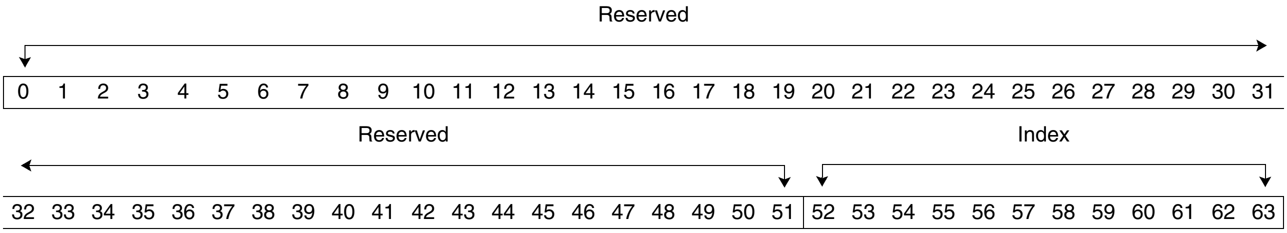
14.2.3 SLB Index Register (SLB_Index)

The SLB Index Register is used to select which SLB entry to access using the SLB_ESID and SLB_VSID registers. The SLB Index is a 12-bit value. The number of index bits used and the organization of the SLB are implementation-dependent. Some implementation can require software to use a specific set of indices for a given SLB value. Refer to the specific implementation documentation for more information.

This register can be written using a single 64-bit store operation or a single 32-bit store operations to the lower 32-bits of this register (that is, offset x'0110C').

Access Type Write

Base Address Offset (BP_Base | P2(n)) + x‘01108’, where n is the SPE number.



Bits	Field Name	Description
0:51	Reserved	Set to zeros.
52:63	Index	Index. The number of bits in this field is implementation-dependent.

14.2.4 SLB Effective Segment ID Register (SLB_ESID)

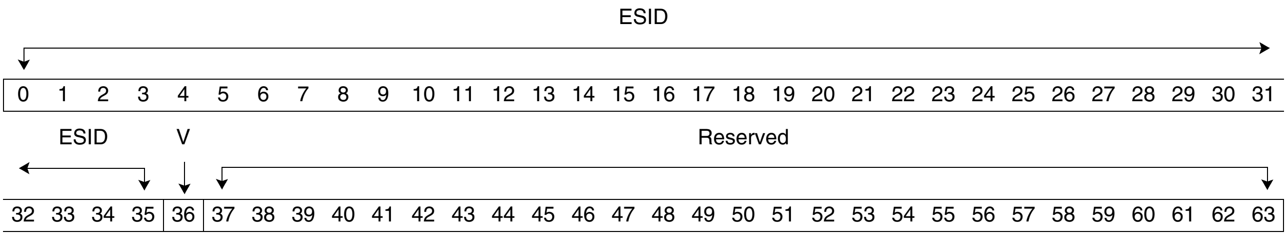
The SLB Effective Segment ID Register is used to access the upper portion of an SLB entry. The SLB_ESID contains the effective segment ID and a bit that indicates if the SLB entry selected by the SLB Index Register is valid.

This register can be written using a single 64-bit store operation or two 32-bit store operations. When using 32-bit operations, the first store must be to the most significant word and the second store must be to the least significant word.

Note: Some implementations can support a cache of the Effective to Real Address Translations (ERATs) to improve performance. Setting the valid bit to ‘0’ does not invalidate any cached translations of the SLB entry. The SLB Invalidate Entry Register must be used for this purpose.

Address Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘01110’, where n is the SPE number.



Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	V	When set, indicates that the SLB entry is valid.
37:63	Reserved	Set to zeros.

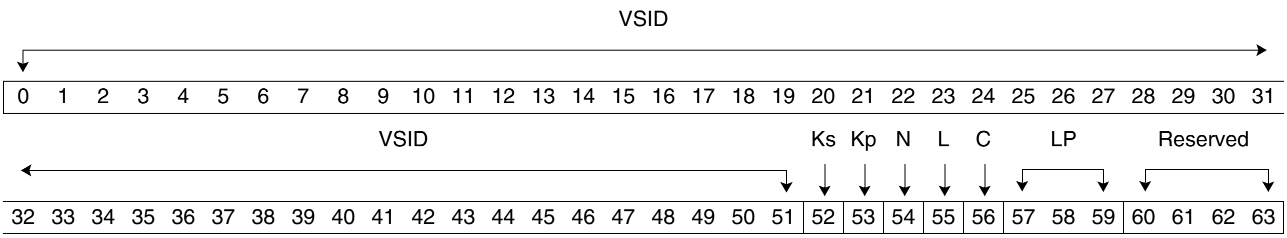
14.2.5 SLB Virtual Segment ID Register (SLB_VSID)

The SLB Virtual Segment ID Register is used to access the lower portion of an SLB entry. The SLB_VSID contains the virtual segment ID and other miscellaneous characteristics of the memory segment. The SLB entry is selected by the SLB Index Register (see page 179).

This register can be written using a single 64-bit store operation or two 32-bit store operations. When using 32-bit operations, the first store must be to the most significant word and the second store must be to the least significant word.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘01118’, where n is the SPE number.



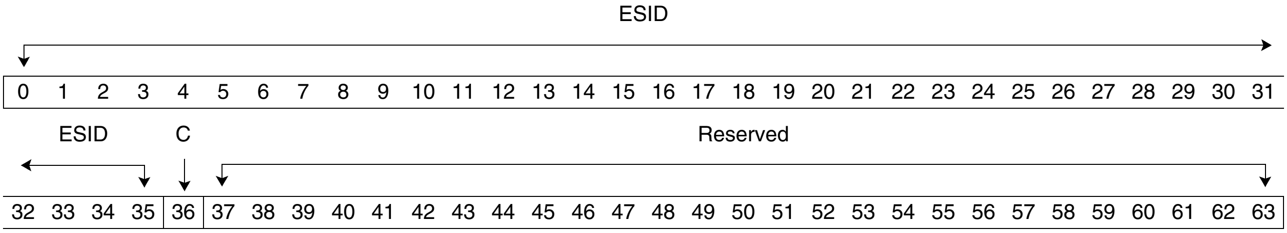
Bits	Field Name	Description
0:51	VSID	Virtual segment ID.
52	Ks	Supervisor (privileged) state storage key This value, along with the Kp bit, is used to compute a key that is used along with the page protection (PP) bits in the PTE for storage protection.
53	Kp	Problem state storage key This value, along with the Ks bit, is used to compute a key that is used along with the PP bits in the PTE for storage protection.
54	N	No execute segment 0 Instruction fetches are allowed. 1 Instruction fetches are not allowed. This is bit is ignored by the MFC.
55	L	Virtual page-size indicator 0 4 KB 1 Virtual pages are large The size of the page is controlled by the LP field in this register.
56	C	Class The class field is used in conjunction with the SLB Invalidate Entry Register (see page 182). It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
57:59	LP	Size selector for large virtual page The size of a page selected by the LP field is implementation-dependent. The number of concurrent page sizes supported is also implementation-dependent. If an implementation supports more page sizes than can be concurrently supported by the LP field, a configuration mechanism should be provided to assign a page size to an LP value. Since this configuration should not change frequently, the configuration mechanism does not need to be directly accessible by privileged software.
60:63	Reserved	Set to zeros.

14.2.6 SLB Invalidate Entry Register (SLB_Invalidate_Entry)

This register can be written using a single 64-bit store operation or two 32-bit store operations. When using 32-bit operations, the first store must be to the most significant word and the second store must be to the least significant word. Any write to the least-significant word causes an entry in the SLB to be invalidated. Writing only the least-significant word will invalidate an entry that matches the upper bits of the SLB_ESID value (contained in the most-significant word) concatenated with the lower bits of the SLB_ESID (provided by data written to the least-significant word).

Address Write

Base Address Offset (BP_Base | P2(n)) + x‘01120’, where n is the SPE number.



Bits	Field Name	Description
0:35	ESID	Effective segment ID.
36	C	Class. The class field is used in conjunction with the SLB Invalidate Entry Register. It is used as an additional qualifier for the ESID when multiple virtual address spaces exist.
37:63	Reserved	Set to zeros.

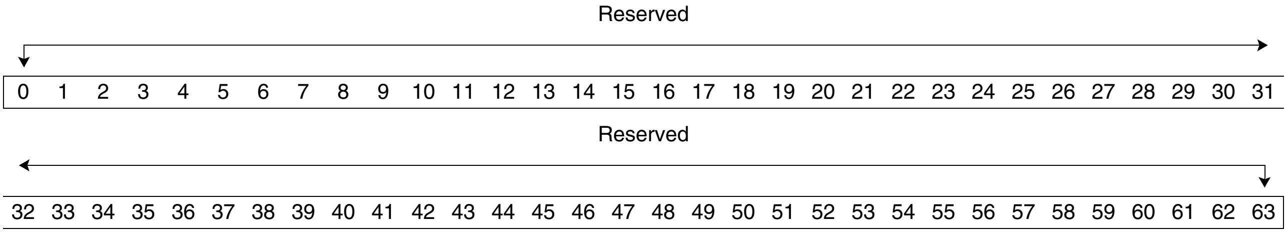
14.2.7 SLB Invalidate All Register (SLB_Invalidate_All)

A write to the SLB Invalidate All Register causes the V bit in all entries of the SLB to be set to ‘0’, making the entry invalid. The remaining fields of each entry are undefined.

This register can be written using a single 64-bit store operation or two 32-bit store operations. Any write to the least-significant word causes an entry in the SLB to be invalidated.

Access Type Write

Base Address Offset (BP_Base | P2(n)) + x‘01128’, where n is the SPE number.



Bits	Field Name	Description
0:63	Reserved	Set to zeros. The data for this register is reserved for future use. Writing to this register causes the contents of the segment lookaside buffer to be voided.

Implementation Note:

The SLB Invalidate All Register for the SPE MMU must not preserve the state of SLB entry zero. This differs from the PowerPC **slbia** instruction. The **slbia** instruction does not alter entry zero.

14.3 Translation Lookaside Buffer Management

The CBEA permits both a hardware TLB reload and a software TLB reload when a translation is not found in the translation lookaside buffer table. For the SPE, the mode is selected by the TL bit of MFC State Register One (see page 197). For the PPE, the mode is selected by bit 52 of LPCR Register. For more information, see *PPE Book IV*.

One difference between a hardware TLB reload and a software TLB reload is the point at which the data storage interrupt (DSI) or SPE interrupt is presented to the PPE. For a software TLB reload, the interrupt is generated when a translation is not found in the TLB. For a hardware TLB reload, the interrupt is generated only after the page table is searched by hardware and a translation is not found. The TLB management facilities function identically for hardware- and software-managed TLB reloads.

Implementation Note:

For a software TLB reload, the implementation can choose the behavior when a translation is not found in the TLB as a result of unset reference or an unset change bit. If an interrupt occurs, then the hardware should simply set these bits in the TLB regardless of the settings of the bits in the software-managed page table.

The hardware-accessed page table is a variably sized data structure that specifies the mapping between virtual page numbers and real page numbers. Each page table entry (PTE) maps one virtual page number to one real page number. The hardware-accessed page table search is defined in *PowerPC Architecture, Book III*. If the translation is not found in the hardware-accessed page table because of a page or mapping fault, a data storage interrupt (DSI) for the PPE or an interrupt for the SPE is posted to the PPE.

In the software TLB reload mode, the format of the page table is software dependent and is not accessed by hardware. When a DSI or an SPE interrupt is posted indicating a translation is not available in the TLB, software should perform a page table search to resolve the TLB miss. In this mode, the MFC Storage Description Register is not used.

For the SPE TLB, a switch between software and hardware management is made by setting the TL bit in MFC State Register One (see page 197). To switch SPE TLB management modes, software should suspend the MFC command queue operation and issue a **sync** instruction before the switch. Switching modes in the PPE is implementation dependent. The registers defined in this section provide the same function for both the PPE and the SPE. The PPE registers are located in the SPR space (see *Table C-1* on page 283), and the SPE registers are located in the CBEA memory map (see *Table A-2* on page 271). The effective address of the operation that causes a translation fault in the PPE is placed in the PPE Data-Address Register, defined in the PowerPC Architecture. The effective address of the operation that causes a translation fault in the MFC is placed in the MFC Data Address Register (see page 201). The MFC Data Address Register is independent for each MFC and the PPE Data Address Register is independent for each PPE.

14.3.1 TLB Mapping

The MFC TLB area must be marked caching inhibited and guarded, and should be restricted to privileged mode code.

In software TLB management mode, hardware must set the TLB Index Hint Register (see page 186) to the TLB entry selected for replacement. Software can change the selected entry as long as the new entry is within the same congruency class. Software must perform the following sequence of operations to replace a TLB entry:

1. Set the index of the TLB entry to be replaced.
2. Set the new contents for the real page number (RPN) portion of the TLB entry.

   Note: A write to the RPN portion of the TLB does not cause the TLB entry to be updated.
3. Set the new contents for the virtual page number (VPN) portion of the TLB entry along with the valid bit.

   Note: A write to the VPN portion of the TLB causes the TLB entry to be updated.

Programming Note:

Depending upon the implementation, invalidating the TLB entry using a **tlbie** (local) instruction, or the TLB Invalidate Entry Register (see page 190) can cause more than one TLB entry to be invalidated (a congruency class, for example). When the TLB is software managed, privileged software must keep a shadow copy of the state of the TLB table, and must reenable the entries that were not meant to be invalidated.

The size and format of the TLB is implementation-dependent. The TLB contents are described in the specific implementation documentation.

14.3.2 TLB Index Hint Register (TLB_Index_Hint)

Note: This register is only used in software management mode. This register is a SPR in the PPE and is accessible using an **mfspr** instruction. This register is mapped into MMIO space for the SPE.

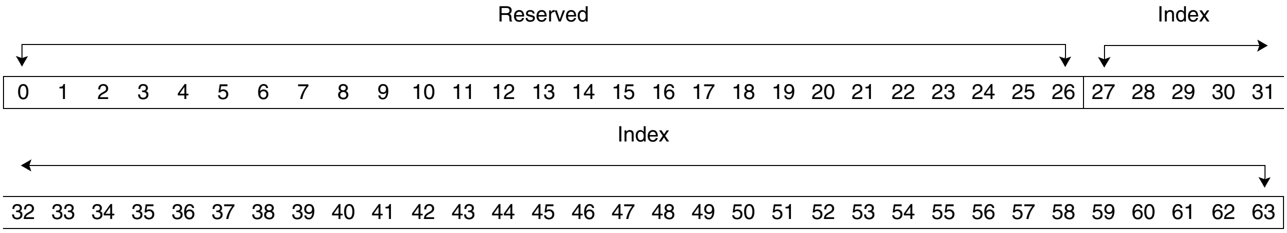
Hardware must set the TLB Index Hint Register to the TLB entry selected for replacement when a translation fault occurs. When software manages the TLB, the TLB Index Hint Register is used to determine which TLB entry the hardware algorithm selected for replacement. Software can read the TLB Index Hint Register and use this value as the TLB Index Register, or it can select a new index within the same congruency class. The TLB Index Hint Register is useful since software cannot always determine the best entry for replacement (that is, the least recently used). Refer to the specific implementation documentation for more information on the TLB Index Hint Register.

The TLB Index Hint Register is a separate register from the TLB Index Register (see page 187) to prevent hardware from changing the index if a fault occurs while software is updating a TLB entry.

Access Type Read Only

Base Address Offset BP_Base | P1(n)) + x‘0500’, where n is the SPE number.

PPE SPR Number x‘3B2’



Bits	Field Name	Description
0:26	Reserved	Set to zeros by hardware.
27:63	Index	Index. This field contains information used by an implementation to update a TLB entry. The TLB index in the index field is a function of the virtual address. The function is implementation-dependent. The number of bits in this field is also implementation-dependent.

14.3.3 TLB Index Register (TLB_Index)

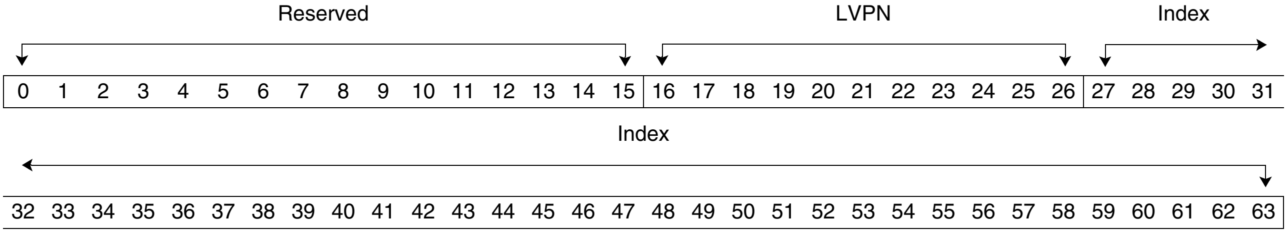
Note: This register is only used in software management mode. This register is an SPR in the PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for the SPE.

The TLB Index Register is used to select the entry in the TLB that will be modified by the TLB Virtual Page Number Register (see page 188) and the TLB Real Page Number Register (see page 189).

Access Type Write

Base Address Offset (BP_Base | P1(n)) + x‘0508’, where n is the SPE number.

PPE SPR Number x‘3B3’



Bits	Field Name	Description
0:15	Reserved	Set to zeros.
16:26	LVPN	Lower virtual page number. These are the lower 11 bits of the virtual page number (the abbreviated VPN from the TLB Virtual Page Number Register (see page 188) concatenated with the LPVN yields the VPN).
27:63	Index	This field contains information used by an implementation to update a TLB entry. The TLB index in the index field is a function of the virtual address. The function is implementation-dependent. The number of bits in this field is also implementation-dependent.

14.3.4 TLB Virtual Page Number Register (TLB_VPN)

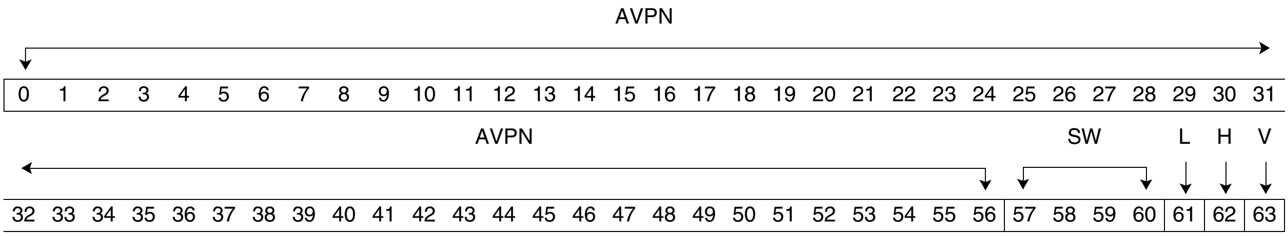
Note: This register is only used in software management mode. This register is an SPR in the PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for the SPE.

The TLB is a cache for the PTE. The TLB Virtual Page Number Register provides access to the upper 64 bits (or the virtual page number portion) of the TLB entry. For more information, see the description of the page table entry in *PowerPC Architecture, Book III.*

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x'0510', where n is the SPE number.

PPE SPR Number x'3B5'



Bits	Field Name	Description
0:56	AVPN	Abbreviated virtual page number.
57:60	SW	An optional field available for software use.
61	L	Virtual page-size indicator. 0 4 KB 1 Virtual pages are large The size of the page is controlled by the LP field in the TLB Real Page Number Register (see page 189).
62	H	Hash function identifier. 0 Primary hash 1 Secondary hash
63	V	0 Entry invalid 1 Entry valid

Note: If the VPN is invalidated to change the protection attributes of a page, or to "steal" a page, a TLB invalid entry command must be issued to invalidate any cache of the effective-to-real-address translation that can be associated with the TLB entry being invalidated. The invalidation selector (IS) field in the TLB invalidate entry command can be used to invalidate the cache without affecting TLB entries.

14.3.5 TLB Real Page Number Register (TLB_RPN)

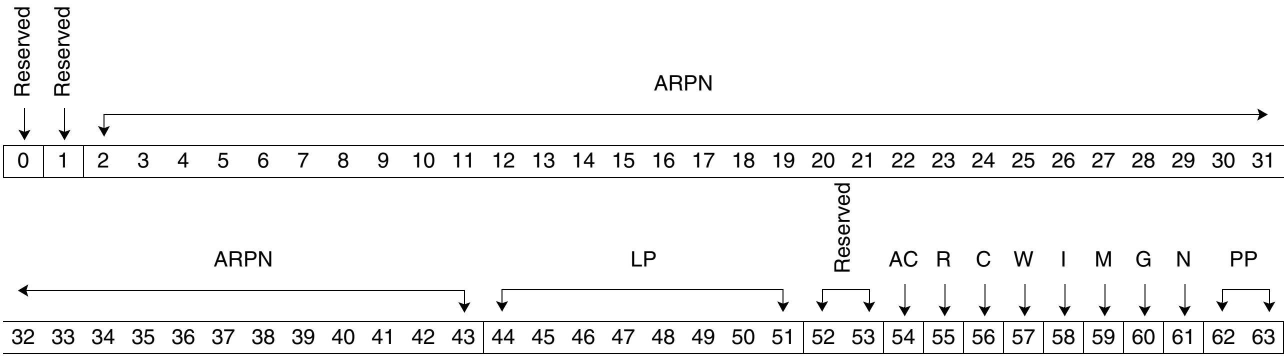
Note: This register is only used in software management mode. This register is an SPR in the PPE and is accessible using either an **mfspr** or an **mtspr** instruction. This register is mapped into MMIO space for the SPE.

The translation lookaside buffer (TLB) is a cache for the page table entry (PTE). The TLB Real Page Number Register provides access to the lower 64-bits (or the real page number portion) of the TLB entry. For more information, see the description of the page table entry in *PowerPC Architecture, Book III*.

Access Type Read/Write

Base Address Offset (BP_Base I P1(n)) + x‘0518’, where n is the SPE number.

PPE SPR Number x‘3B4’



Bits	Field Name	Description
0	Reserved	Set to zeros.
1	Reserved	Set to zeros.
2:43	ARPN	Abbreviated real page number
44:51	LP	Size selector for a large virtual page. This field is used to select the size of a large page from the hardware-defined list of page sizes. This field supports up to eight concurrent large page sizes. The size of a page selected by the LP field is implementation-dependent. The number of concurrent page sizes supported is also implementation-dependent. The least significant bit of this field is used as part of the comparison to the virtual address on a TLB lookup. Depending on the page size, some bits of this field may concatenated with the ARPN field to form the RPN for a successful lookup. **aaaaaaaa** — TLB_Invalidate_Entry[L] = ‘0’. **aaaaaaa0** —Large page size one if MFC_TLB_VPN[L] = '1'. **aaaaaa01** — Large page size two if MFC_TLB_VPN[L] = '1'. ... **01111111** — Large page size eight if MFC_TLB_VPN[L] = '1'.
52:53	Reserved	Set to zeros.
54	AC	Address compare bit. 0 Data address compare disabled for the corresponding virtual page. 1 Data address compare enabled for the corresponding virtual page.
55	R	Reference bit. The effects of this bit in software management mode is implementation-dependent.
56	C	Change bit. The effects of this bit in software management mode is implementation-dependent.
57	W	Write through storage control bit.
58	I	Caching inhibit storage control bit.

Bits	Field Name	Description
59	M	Memory coherency storage control bit.
60	G	Guarded storage control bit.
61	N	No execute page if N = ‘1’.
62:63	PP	Page protection bits.

14.3.6 TLB Invalidate Entry Register (TLB_Invalidate_Entry)

Note: This register is only used in the software management mode, or when privileged software preloads a TLB. This register is not available for a PPE. The PPE uses a **tlbie** or **tlbiel** instruction to invalidate TLBs. This register is mapped into MMIO space for the SPE.

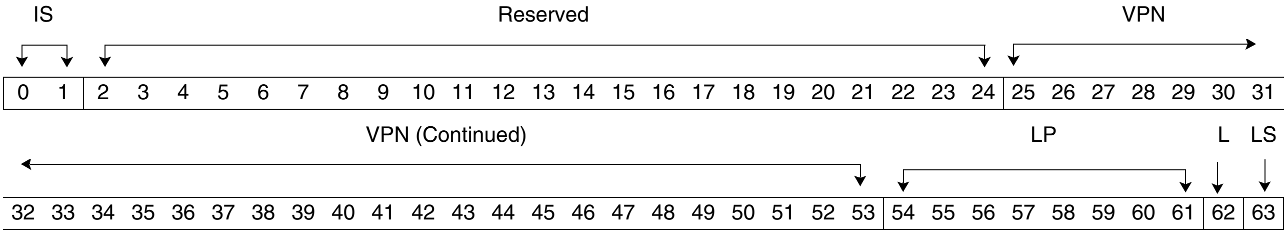
The TLB Invalidate Entry Register is used to invalidate TLB entries in the MFC. The function of this register is similar to the PowerPC **tlbie** instruction. Access to this register is privileged.

The TLB Invalidate Entry Register contains a virtual page number (VPN) field and an invalidation selector (IS) field. The VPN is used to identify a particular entry to invalidate, and the IS field is used to control how selective the invalidate should be.

Access Type Write

Base Address Offset (BP_Base | P1(n)) + x‘0540’, where n is the SPE number.

PPE SPR Number No corresponding SPR number exists. The PPE uses the local form of the **tlbie** instruction.



Bits	Field Name	Description
0:1	IS	Invalidation selector. 00 The TLB is as selective as possible in invalidating the TLB entry. The implementation should use as many VPN bits as possible to eliminate invalidating unnecessary entries. 01 The TLB entry is not invalidated. Any lower-level caches of the translation are invalidated. 10 The TLB does a congruency-class invalidate if the LPID matches the current value in the MFC Logical Partition ID Register (see page 199). 11 The TLB does a congruency-class invalidate regardless of LPID match. **Implementation Note**: An implementation can choose to implement a subset of these options. It is always acceptable for an implementation to invalidate more TLB entries than specified by this instruction. The IS bits are only a useful hint for a performance benefit.
2:24	Reserved	Set to zeros.
25:53	VPN	Bits 32-59 of the virtual address.

Bits	Field Name	Description
54:61	LP	Size selector for a large virtual page. This field is used to select the size of the large page from the hardware-defined list of page sizes. The size of a page selected by the LP field is implementation-dependent. The number of concurrent page sizes supported is also implementation-dependent. This field supports two concurrent large page sizes. This LP field selects one of those two pages. Depending on the page size selected, some bits in this field may be concatenated with the VPN field to determine which entries are invalidated. **aaaaaaaa** — TLB_Invalidate_Entry[L] = ‘0’. **aaaaaaa0** — IF TLB_Invalidate_Entry[L] = '1', large page size one selected. **aaaaaa01** — IF TLB_Invalidate_Entry[L] = '1', large page size two selected. ... **01111111** — Large page size eight if TLB_Invalidate_Entry[L] = '1'. Software should set the least-significant bit of the LP field to the same value as the LS bit for compatibility with implementations that only support two large page sizes.
62	L	Large Page indicator 0 Page is small (4KB) 1 Page is large. See LP field or local storage put (Lp) bit in the MFC Address Compare Control Register (see page 203).
63	LS	Large Page Selection 0 First large page (The size is implementation-dependent.) 1 Second large page (The size is implementation-dependent.) **Note:** Software should set this bit to the same value as the least-significant bit of the LP field for compatibility with implementations that only support two large page sizes.

Programming Note:

1. If the VPN is being invalidated to change the protection attributes of a page, or to “steal” a page, a TLB Invalidate Entry command must be issued to invalidate any cache of the effective-to-real-address translation that can be associated with the TLB entry being invalidated. The IS field in the TLB Invalidate Entry command is used to invalidate the cache without affecting TLB entries.
2. Care must be taken in using this function in real-time or TLB managed environments, since hardware can invalidate all TLB entries in the associated congruency class. This could adversely affect TLB set management and real-time deterministic response. To avoid this side effect for real-time environments, privileged software can use the TLB index and TLB direct modification functions to locate the specific entry to be invalidated in the congruency class and only invalidate the entry that matches.

14.3.7 TLB Invalidate All Register (TLB_Invalidate_All)

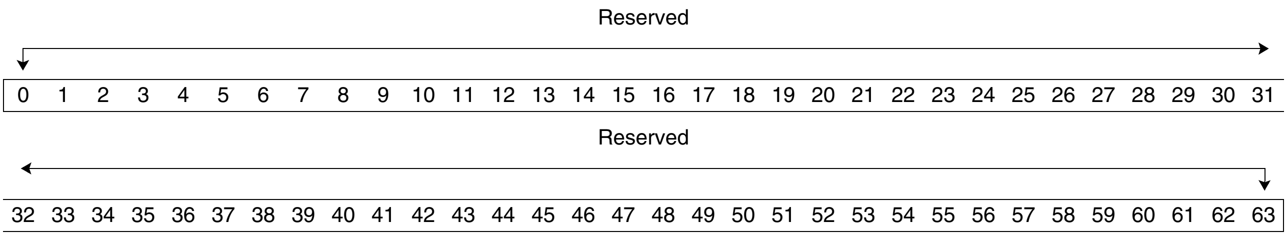
Note: This register is only used in the software management mode, or when privileged software preloads a TLB. This register is not available for a PPE. The PPE uses a **tlbia** instruction to invalidate TLBs. This register is mapped into MMIO space for the SPE.

The TLB Invalidate All Register is used to invalidate all TLB entries in the MFC command queues. The function of this register is similar to the PowerPC **tlbia** instruction. Access to this register is privileged.

Access Type Write

Base Address Offset (BP_Base | P1(n)) + x‘0548’, where n is the SPE number.

PPE SPR Number No corresponding SPR number exists. The PPE uses the local form of the **tlbia** instruction.



Bits	Field Name	Description
0:63	Reserved	Set to zeros. The data for this register is reserved for future use. The writing to this register causes the contents of the TLB to be voided.

14.4 Real-Mode Storage Control Facilities (Optional)

14.4.1 PPE Real-Mode Storage Control Facility

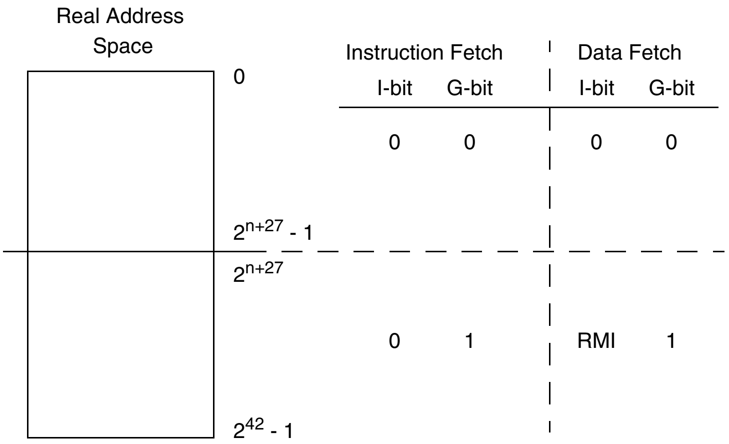
The PPE Real-Mode Storage Control Facility in the PowerPC processor is an optional facility in the PowerPC Architecture.

The PPE supports a 4-bit real-mode storage control (RMSC) field in an implementation-dependent register. The RMSC field is used to control the guarded storage control attribute when the PPE is running with translation off (that is, PowerPC MSR[IR] = ‘0’ and MSR[DR] = ‘0’). Access to this facility is privileged. The RMSC field provides a mechanism for setting the boundary between storage that is considered well behaved and storage that is not. The granularity of this boundary, illustrated in *Figure 14-1* is 256 MB. The RMSC field has no effect when the PPE is running with translation on (that is, PowerPC MSR[IR] = ‘1’ and MSR[DR] = ‘1’).

If the RMSC field is set to a value of ‘n,’ all accesses within the first $2^{(n+27)}$ bytes, for $1 \leq n \leq 15$ of the real address space are considered well behaved and cacheable. Memory that is well behaved typically has the guarded (G) attribute set to zero. The caching inhibited (I) attribute can be either ‘0’ or ‘1’, but in real mode is typically set to ‘0’. Data accesses outside the first $2^{(n+27)}$ bytes of the real address space can be neither well behaved nor cacheable. In this case, the guarded (G) bit is set to ‘1’. Caching is controlled by the RMI bit in an implementation-dependent PowerPC register in real addressing mode.

When the RMSC field is set to a value of ‘0’, no data accesses are considered well behaved and caching is controlled by the RMI bit (that is, I = RMI, G = ‘1’) in real addressing mode. All instruction fetches are not well behaved, but are cacheable (that is, the caching inhibited attribute, I, equals zero, and the guarded attribute, G, equals ‘1’) in real addressing mode.

Figure 14-1. Real-Mode Storage Boundary (showing Instruction Fetches and Data Fetches)



14.4.2 MFC Real-Mode Address Boundary Register MFC_RMAB)

The MFC Real-Mode Address Boundary Facility in the MFC is an optional facility in the CBEA. The MFC supports a 4-bit real-mode boundary (RMB) field in this implementation-dependent register.

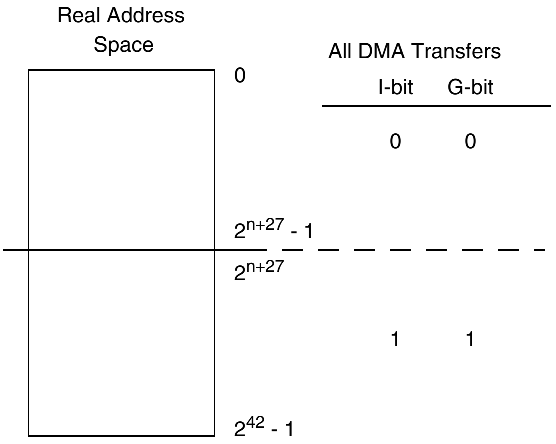
The RMB field is used to control the caching inhibited (I) and guarded (G) storage control attributes, when the MFC is running with translation off, which means that real-mode addressing is on, MFC_SR1[R] = 0 of the MFC State Register One (see page 197). The 4 lower bits of this register provide a mechanism for setting the boundary between storage that is considered well behaved and cacheable and storage that is not. The granularity of this boundary, illustrated in *Figure 14-2*, is 256 MB.

The RMB field has no effect when the MFC is running with translation on, that is, when the Relocate (R) bit is set to '1' in the MFC State Register One (see page 197).

For this operation to work properly, privileged software must suspend all MFC operations before modifying the contents of this register.

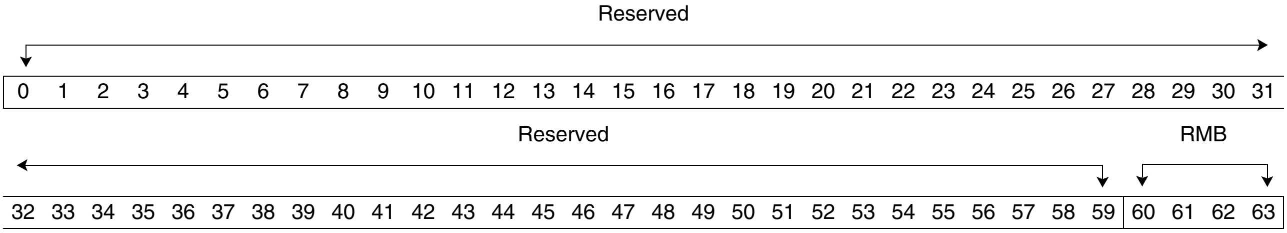
The MFC Real-Mode Address Boundary Register is an implementation-dependent register. Access to this register is privileged.

Figure 14-2. Real-Mode Storage Boundary (showing all DMA transfers)



Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘0900’, where n is the SPE number.



Bits	Field Name	Description
0:59	Reserved	Set to zeros.
60:63	RMB	Real-mode boundary. If this field is set to a value of ‘n’, only those accesses within the first 2^{n+27} bytes for 1 ≤ n ≤ 15 of the real address space are considered well behaved and cacheable in real addressing mode (MFC_SR1[R] = 0). All accesses outside the first 2^{n+27} bytes of the real address space are neither well behaved nor cacheable in real addressing mode. If the RMB field is set to a value of x‘0’, no accesses are considered well behaved and cacheable in real addressing mode.

15. MFC Privileged Facilities

The registers in these sections are only accessed by software.

The MFC Privilege 1 Facilities include the following registers:

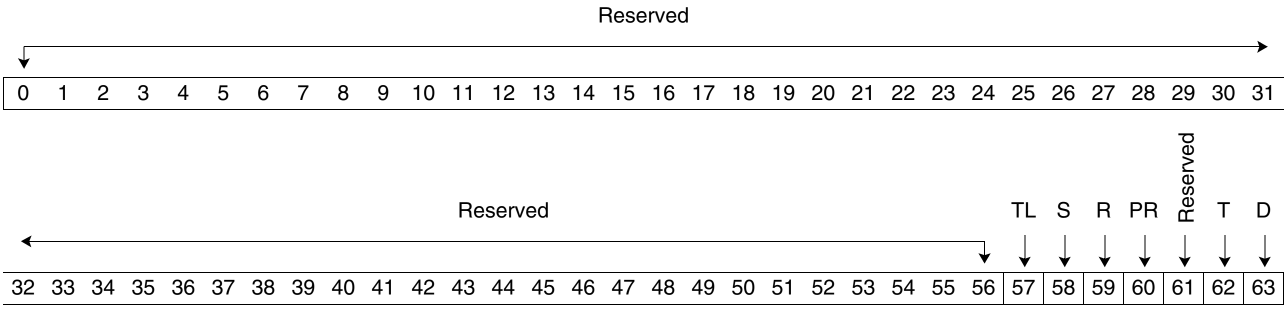
- MFC State Register One (see below)
- MFC Logical Partition ID Register (see page 199)
- MFC Storage Description Register (see page 200)
- MFC Data Address Register (see page 201)
- MFC Data Storage Interrupt Status Register (see page 202)
- MFC Address Compare Control Register (see page 203)
- MFC Local Storage Address Compare Facility (see page 205)
- MFC Command Error Register (see page 207)
- MFC Data Storage Interrupt Pointer Register (see page 208)
- MFC Control Register (see page 209)
- MFC Atomic Flush Register (see page 212)
- SPU Outbound Interrupt Mailbox Register (see page 213).

15.1 MFC State Register One (MFC_SR1)

MFC State Register One contains configuration information controlled by a hypervisor. Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0000’; where n is the SPE number



Bits	Field Name	Description
0:56	Reserved	Reserved.
57	TL	Software or hardware page table search 0 TLB miss is handled by hardware (SLB and PTE misses are always handled by software). 1 TLB miss is handled by software.

Bits	Field Name	Description
58	S	SPU master run control. Setting this bit to zero suspends the dispatch of instructions in the SPU. The state of this bit does not affect the state of the SPU Run Status in the SPU Status Register (see page 87). 0 SPU is stopped. 1 SPU is controlled by the SPU Run Control Register (see page 86). **Programming Note:** The SPU master run control (S) bit allows privileged software to keep an SPU from running even if an application set the SPU run bit in the SPU Run Control Register (see page 86). This bit is useful during power savings mode.
59	R	Relocate. 0 MFC translation off 1 MFC translation on **Programming Note:** The MFC relocate (R) bit controls how effective addresses in MFC commands are translated into real addresses. If the relocate control specifies translation off (R = ‘0’), the effective addresses used in MFC commands are real addresses and are subject to the real-mode offset control facility. If the relocate control is enabled (R = ‘1’), the effective addresses used in MFC commands are translated. The SLB, TLB, and page table facilities are used to translate the effective address to a virtual address, and then to a real address. With translation on, the real-mode offset control is not involved.
60	PR	Problem state. 0 The MFC has privileged state access to pages. 1 The MFC has problem state access to pages. **Programming Note:** The MFC problem state (PR) bit is set by privileged software based on the use of the associated SPU. If the SPU is to be a privileged software resource (not under direct control of an application). and the function requires the SPU to issue MFC commands with privileged state access to pages, this bit should be cleared. If the SPU function is controlled by an application, its MFC access should be restricted to problem state via the PR bit. The problem state control bit interacts with the Ks and Kp storage key bits in the SLB in combination with the page protection (PP) bits in the page table, as defined in PowerPC Architecture, Book III. The problem state control is only effective in MFC translation on state (R = ‘1’).
61	Reserved	Reserved.
62	T	Bus **tlbie** enable. 0 Ignore **tlbie** commands on the bus. 1 Invalidate TLB entries in response to a **tlbie** on the bus. **Programming Note:** The **tlbie** (T) bit is typically enabled if page tables are used by multiple PPEs and MFCs. If both **tlbie**-managed and MFC managed TLBs are enabled, the MFC will participate in broadcast **tlbie** operations (initiated by the PPE **tlbie** instruction). If disabled, the MFC will ignore the **tlbie** broadcast. This state is useful when each MFC uses its own private page table.
63	D	Local Storage real address decode. 0 Disable system real address of local storage. The MFC does not decode the local storage area for the associated SPU. All accesses are ignored. 1 Enable system real address of local storage The MFC decodes the local storage area for the associated SPU. All accesses are performed. **Programming Note:** Privileged software can use the local storage real address decode enable/disable (D) bit to isolate the local storage from being addressed physically (through the page table, or with I/O devices on the Element Interconnect Bus (EIB). This can be done to save power or to facilitate a more isolated SPU program environment. Note that local storage access with the local storage address in MFC DMA commands and SPU load-and-store from local storage is not affected by this setting.

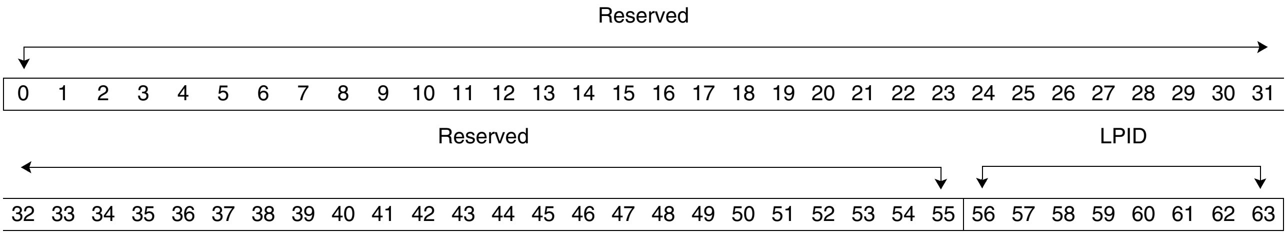
15.2 MFC Logical Partition ID Register (MFC_LPID)

The PowerPC Architecture provides a logical partitioning (LPAR) facility to permit processors and portions of main storage to be allocated to logical groups or partitions. For more information on the LPAR, see *PowerPC Architecture, Book III*.

The CBEA extends the LPAR facility to allow SPEs to also be assigned to partitions. The MFC Logical Partition ID Register contains a value that identifies to which partition an SPE is assigned.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0008’; where n is the SPE number



Bits	Field Name	Description
0:55	Reserved	Reserved
56:63	LPID	Logical partition ID.

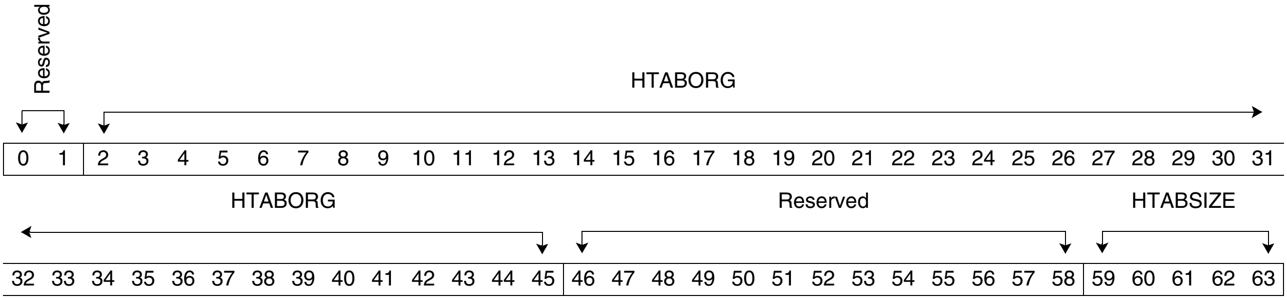
15.3 MFC Storage Description Register (MFC_SDR)

The MFC Storage Description Register contains the starting address in main storage and the size of the page table for the associated MFC. The MFC Storage Description Register provides the same function as the PowerPC Storage Description Register (SDR). For more information on the SDR, see *PowerPC Architecture, Book III*.

When the SPE is configured for software TLB management (that is, MFC_SDR[TL] = '1'), the value in this register is not used.

Access Type Read/Write

Base Address Offset (BP_Base I P1(n)) + x‘0400’; where n is the SPE number



Bits	Field Name	Description
0:1	Reserved	Reserved.
2:45	HTABORG	Page table origin (real address of the page table). The HTABORG field in SDR1 contains the high-order 44 bits of the 62-bit real address of the page table. The page table is thus constrained to lie on a 2^{18} byte (256 KB) boundary at a minimum. The number of low-order zero bits in HTABORG must be greater than or equal to the value in HTABSIZE. On implementations that support a real address size of only m bits, where m is less than 62, the upper bits of the page table origin are treated as reserved bits, and software must set them to zeros.
46:58	Reserved	
59:63	HTABSIZE	Encoded size of page table. The HTABSIZE field in SDR1 contains an integer giving the number of bits (in addition to the minimum of 11 bits) from the hash that are used in the page table index. This number must not exceed 28.

15.4 MFC Data Address Register (MFC_DAR)

The MFC Data Address Register contains the effective address associated with an MFC data segment interrupt or an MFC data storage interrupt. The function of this register is similar to the PowerPC DAR. For more information on the DAR, see *PowerPC Architecture, Book III.* Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0620’; where n is the SPE number

Effective Address																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Effective Address																															
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

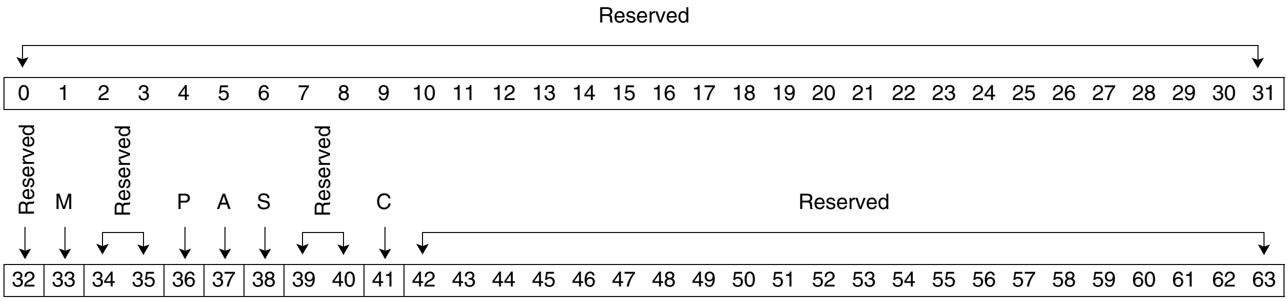
Bits	Field Name	Description
0:63	Effective Address	Effective address associated with the data segment or data storage interrupt.

15.5 MFC Data Storage Interrupt Status Register (MFC_DSISR)

The MFC Data Storage Interrupt Status Register contains the status that defines the cause of the MFC data storage interrupt. The function of this register is similar to the PowerPC DSISR. For more information, see *PowerPC Architecture, Book III*. Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0610’; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Reserved	Set to ‘0’.
33	M	Set to ‘1’ if the PTE is not found or a TLB miss occurs while in software-managed TLB mode.
34:35	Reserved	Set to zeros.
36	P	Set to ‘1’ if the access is not permitted by the storage protection mechanism.
37	A	Set to ‘1’ if the atomic access is to a write through or caching inhibited page.
38	S	Set to ‘1’ if the access was a **put*[rlfs]***, a **putll*[u]*c**, or a **sdcrz** operation.
39:40	Reserved	Set to zeros.
41	C	Set to ‘1’ if a data address compare match occurs (all DMA issue activity is halted).
42:63	Reserved	Set to zeros.

15.6 MFC Address Compare Control Register (MFC_ACCR)

The MFC data address compare mechanism allows the detection of DMA access to a virtual page marked with the address compare (AC) bit in the page table entry (PTE) set and a range within the local storage. This facility is normally used for debug. This debug mechanism is controlled by the MFC Address Compare Control Register for both effective address and local storage address compares. For effective address compares, the AC bit in the page table entry (PTEAC) controls which pages are included in the comparison. For local storage compares, the MFC Local Storage Address Compare Facility (see page 205), contains the address used for the comparison and a mask indicating which address bits should be included in the comparison.

An effective address compare match occurs for a **put**, **get**, **getllar**, **putll[u]c**, or **sdcrz** operation if the following conditions are true for any byte accessed (including effective addresses used in list commands):

- The PTE address compare (AC) bit is set, the operation is a **put**, and the MFC_ACCR put (Ep) bit is set.
- The PTE address compare (AC) bit is set, the operation is a **get**, and the MFC_ACCR get (Eg) bit is set.

A local storage address compare match occurs for a **put**, **get**, **getllar**, **putllc**, and **putlluc** operation if, for any byte access, the following conditions are true (including local storage addresses accessed by list commands). The local storage address value compare occurs before the SPU Local Storage Limit Register (see page 217) wrap (if any) is applied.

- The operation is a **put**, the local storage address being read matches the address range specified by the bit-wise AND of the local storage compare address mask and the local storage compare address in the MFC Local Storage Address Compare Facility (see page 205), and the local storage read bit (Lp) in the MFC_ACCR is set.
- The operation is a **get**, the local storage address being written matches the address range specified by the bit-wise AND of the local storage compare address mask and the local storage compare address in the MFC Local Storage Address Compare Facility (see page 205), and the local storage get bit (Lg) in the MFC_ACCR is set.

Once an PTE address compare match occurs, an MFC data storage interrupt is presented, and the C bit is set in the MFC Data Storage Interrupt Status Register (see page 202). Once a local storage address compare match occurs, a class 1 interrupt is presented, and the LP or LG bits are set in the Class 1 Interrupt Status Register (see page 258). For both PTE and local storage compares, all MFC DMA operations are stopped at the command that caused the compare match. Some portion or all of the command might have been completed before the stop. Due to the weakly ordered model, additional commands can also have been started when the address compare occurs. For PTE address compares, the MFC Data Storage Interrupt Pointer Register (see page 208), will contain the index of the command that triggered the address compare, and the MFC_DAR will contain the effective address of the access that triggered the compare condition.

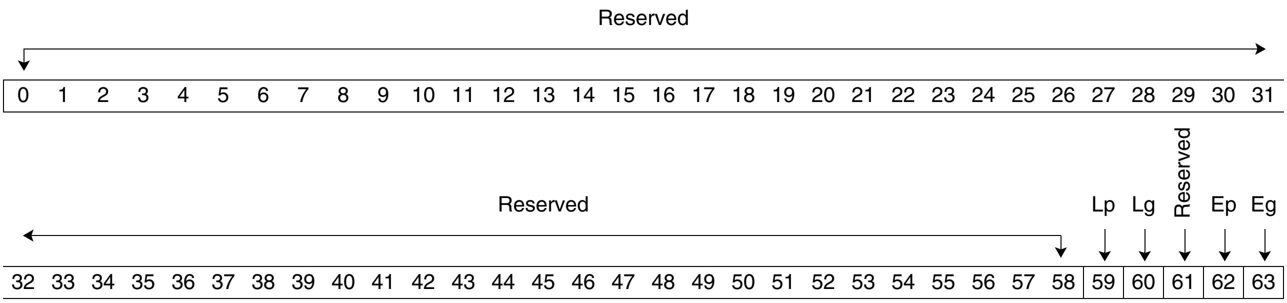
For local storage address compare matches, the MFC_LSCRR contains the local storage address of the access that triggered the compare condition, and the index of the command that triggered the address compare.

For both local storage and PTE address compare stops, the DMA operations can be resumed by writing the MFC Control Register with the Sc bit cleared, which indicates resume normal operation. In addition to setting the Sc bit to zero, the AC bit in the PTE must be reset and the R bit in the MFC Control Register must also be set to resume a command stopped due to a PTE address compare. Setting the R bit causes the MFC to re-translate the command.

Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0600’; where n is the SPE number



Bits	Field Name	Description
0:58	Reserved	Reserved.
59	Lp	Enable of local storage address for **put** commands (includes **sdcrz** and **putll[u]c** operations).
60	Lg	Enable of local storage address for **get** commands.
61	Reserved	Reserved.
62	Ep	Enable of effective address for **put** commands (includes **sdcrz** and **putll[u]c** operations).
63	Eg	Enable of effective address for **get** commands.

Note: There is no restriction on the setting of enables in this register. See the *PowerPC Architecture, Book III* document for a complete description of the Ep and Eg bits.

15.7 MFC Local Storage Address Compare Facility

The use of this facility is explained in the *Section 15.6 MFC Address Compare Control Register* on page 203. This facility consists of:

- MFC Local Storage Address Compare Register
- MFC Local Storage Compare Result Register (see page 206).

15.7.1 MFC Local Storage Address Compare Register (MFC_LSACR)

The MFC Local Storage Address Compare Facility contains the local storage address and local storage address mask to be used in the MFC local storage address compare operation selected by the MFC Address Compare Control Register (see page 203). Access to this register is privileged.

A local storage address compare occurs when the local storage address (LSA) accessed is within the range of addresses specified by the bit-wise AND of the local storage compare address mask (LSCAM) and the local storage compare address (LSCA) fields.

- Get Match = ((LSCA & LSCAM) == (LSA & LSCAM) & MFC_ACCR[Lg])
- Put Match = ((LSCA & LSCAM) == (LSA & LSCAM) & MFC_ACCR[Lp])

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0808’; where n is the SPE number

LSCAM

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

LSCA

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	LSCAM	Local storage compare address mask. The number of bits implemented in the mask field is implementation-dependent. A ‘0’ value written to a bit in this field makes the corresponding bit in the local storage address a “don’t care” in the MFC local storage address compare operation. A ‘1’ value written to a bit in this field includes the corresponding local storage address bit in the MFC local storage address compare operation.
32:63	LSCA	Local storage compare address. The number of upper bits implemented in the address and mask fields is implementation-dependent.

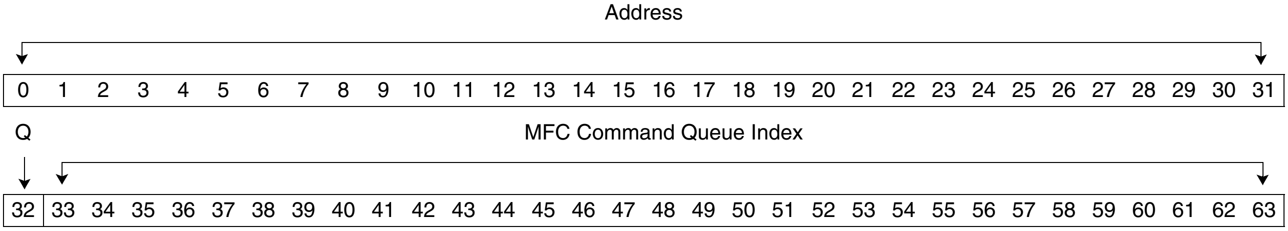
15.7.2 MFC Local Storage Compare Result Register (MFC_LSCRR)

The MFC Local Storage Compare Result Register contains the local storage address that triggered the compare along with the MFC command queue index of the MFC command that triggered the compare stop. Contents of this register are only valid when a Class 1 Interrupt occurs with the Lp or Lg interrupt status bits set. The Q bit indicates if the command was issued from the PPE side or SPU side of the command queue.

Access to this register is privileged. The contents of this register become indeterminate once MFC operation is resumed.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0810’; where n is the SPE number



Bits	Field Name	Description
0:31	Address	Address of access triggering local storage address compare
32	Q	Command queue 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the queue entry that triggered the compare stop.

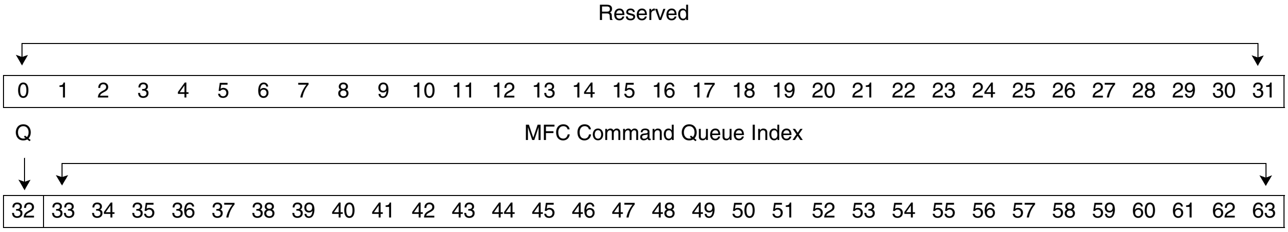
15.8 MFC Command Error Register (MFC_CER)

The MFC Command Error Register contains the index of the command in the MFC command queue associated with an MFC error condition. When an error is detected by the MFC, all processing is suspended until the error has been cleared and the MFC DMA operation restarted. The MFC DMA operation is restarted by writing the MFC Control Register (see page 209), with the MFC restart (R) bit set.

Note: Command errors can occur on the MFC proxy and MFC SPU command queues. The MFC must stop execution on the first error. The MFC Command Error Register must point to the command that caused the first error.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0C00’; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Q	Command queue. 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the queue entry that caused the command error. The number of bits implemented in this field is implementation-dependent.

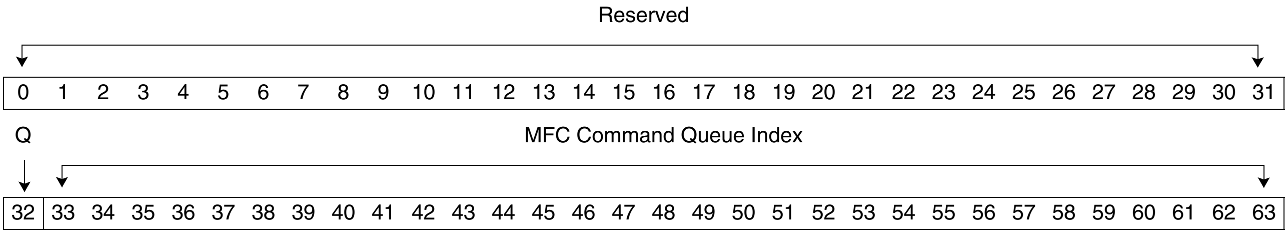
15.9 MFC Data Storage Interrupt Pointer Register (MFC_DSIPR)

The MFC Data Storage Interrupt Pointer Register contains the index for the command in the DMA command queue associated with an MFC data storage interrupt (DSI) or an MFC data segment interrupt.

The cause of an MFC data storage interrupt is identified in the MFC Data Storage Interrupt Status Register (see page 202).

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0800’; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32	Q	Command queue. 0 Index is for the MFC proxy queue. 1 Index is for the MFC SPU queue.
33:63	MFC Command Queue Index	MFC command queue index. Points to the command that caused the data storage interrupt. The number of bits implemented in this field is implementation-dependent.

Implementation Note:

Only one translation fault can be outstanding. The implementation can either stop all MFC command queue processing on the first translation error or continue processing. If processing is continued, all ordering rules must be followed (a command must not be processed if it is dependent on a command that is waiting for a translation fault to be resolved). The state of the MFC must appear as if the command (or partial command) were never issued. This is also the case if a second translation fault occurs.

15.10 MFC Control Register (MFC_CNTL)

The MFC Control Register allows privileged software to govern the operation of the MFC and the MFC commands. There is one register for each SPU within an SPU group.

Setting the suspend control (Sc) bit while the suspend mask (Sm) bit is a 0 causes the MFC to stop executing commands. The suspend control (Sc) bit is ignored if the suspend mask (Sm) bit is a 1 when the MFC Control register is written. MFC commands might still be enqueued while an MFC command queue is suspended. To change the state of the suspend control (Sc) bit, the suspend mask (Sm) bit must be set to zero. If the suspend mask (Sm) is set to one, the state of the suspend bit (Sc) will not be updated. The suspend mask (Sm) bit should be written as a 1 value when the MFC Control register is written with no intent to change the current MFC operation state (suspended/normal).

Setting the purge (Pc) bit in this register causes the MFC to remove all commands from the MFC command queue. Hardware resets this bit when a purge operation completes. MFC commands are not enqueued while the MFC command queue is in the purge state.

Note: The restart (R) bit must not be set to resume an MFC command stopped due to a local storage address compare.

Setting the restart (R) bit causes the MFC command with a pending translation fault to be reissued.
The restart (R) bit is automatically reset by hardware after a pending MFC command is reissued.
The restart operation is only effective if the MFC command queue is in normal queue operational status. Software must set the restart (R) bit to ‘1’ to resume an MFC command either after one of the faults listed below or after a page protection fault has been indicated.

- A fault is either a data segment interrupt or a data storage interrupt with the miss (M) bit set in the MFC Data Storage Interrupt Status Register (see page 202).
- A page protection fault is a data storage interrupt with the protection (P) bit set in the MFC_DSISR.

Software can set the suspend (Sc) bit and the restart (R) bit in the same MMIO write operation.

The decrementer halt (Dh) bit allows privileged software to stop the decrementer. The decrementer remains halted until the Dh bit is reset and the SPU issues a write channel (**wrch**) instruction to the SPU Decrementer (see page 128).

The decrementer status (Ds) bit reflects the state of the decrementer (running or not running), which allows privileged software to determine if the application running in the SPU used the decrementer. The state of the decrementer is required for the context save and resume of the SPE.

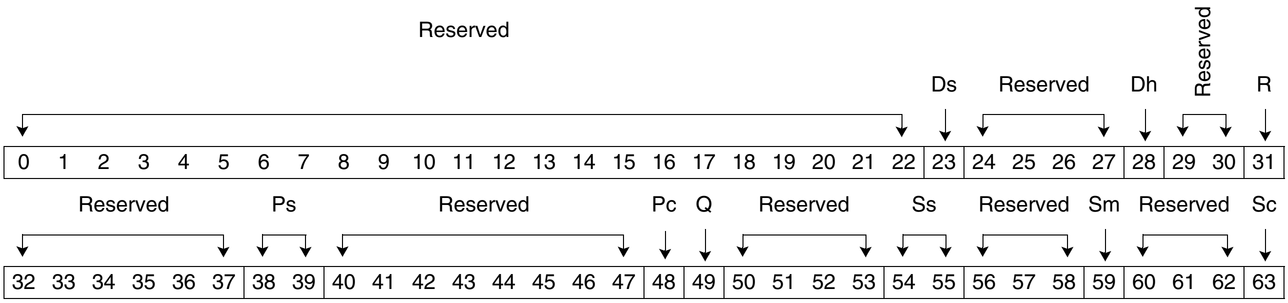
When either an MFC command error, a PTE address compare stop, a local storage address compare, or an MFC hardware error occurs, the suspended MFC command queue operation bit (Sc) is set; MFC operations are suspended for both MFC command queues, and the MFC command queue status changes to MFC command queue operation suspended status (Ss equals ‘11’).

To resume normal MFC command queue operations:

- After a PTE address compare stop, the AC bit in the PTE must be reset, then the MFC Control Register (see page 209), must be written to set the Sc bit to ‘0’ and to reset the restart bit (R) to ‘1’. Resetting the restart bit (R) causes the MFC to re-translate the command.
- After a local storage address compare stop, the MFC Control Register (see page 209), must be written to set the Sc bit to ‘0’. Do not set the restart (R) bit to resume an MFC command stopped due to a local storage address compare.
- After an MFC command or an MFC hardware error, the purge sequence bits (Pc and Ps) should be issued before setting normal MFC command queue operation mode.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘03000’; where n is the SPE number



Bits	Field Name	Description
0:22	Reserved	Reserved.
23	Ds	SPU decrementer status. This is a read only field. Data written to this field will be ignored. 0 Decrementer not running. 1 Decrementer running.
24:27	Reserved	Reserved.
28	Dh	SPU decrementer halt. 0 Decrementer is allowed to run if activated by a write to the Set Decrementer Channel. 1 Decrementer halted.
29:30	Reserved	Reserved.
31	R	Restarts the MFC command that caused the translation fault or a PTE address compare stop. *(This bit must not be set to resume an MFC command stopped due to a local storage address compare. This bit must be set to ‘1’ to restart the MFC command operation that caused a translation fault or a PTE address compare stop. This bit is automatically reset by hardware after the MFC command has been resumed.)* 0 No MFC command restart requested. 1 Write: Restart MFC command. Read: MFC command reissue pending.
32:37	Reserved	Reserved.
38:39	Ps	MFC command queue purge status. This is a read only field. Data written to this field will be ignored. 00 Purge request not outstanding. 01 Purge of MFC command queues in process (some implementations can choose not to implement this state. 11 Purge of MFC command queues is complete.
40:47	Reserved	Reserved.

Bits	Field Name	Description
48	Pc	Purge MFC commands from the MFC SPU command queue and MFC proxy command queue. 0 No purge of MFC command queues requested 1 Purge MFC command queues request.
49	Q	MFC command queue empty status. This is a read only field. Data written to this field is ignored. 0 MFC SPU command queue and MFC proxy command queue are not empty. 1 MFC SPU command queue and MFC proxy command queue are both empty, and all MFC commands are complete.
50:53	Reserved	Reserved.
54:55	Ss	MFC command queues suspend status. This is a read only field. Data written to this field will be ignored. 00 Normal MFC command queues operation. 01 Suspend of MFC command queues in process (some implementations can choose not to implement this state). 11 MFC command queue operation suspended (both queues).
56:58	Reserved	Reserved.
59	Sm	Suspend Mask This bit is used to control the effect of the Suspend Control (Sc) bit when writing the MFC Control Register. This bit is used by system software to avoid unintentional changes to the MFC state. 0 Effect of Sc bit is enabled 1 Effect of Sc bit is disabled
60:62	Reserved	Reserved.
63	Sc	Suspend Control. Suspend MFC command queues operation. 0 Normal MFC command queue operation request (both queues). 1 Suspend MFC command queue operation request (both queues).

Implementation Note:

Once the suspend MFC command queue operation (Sc) bit is set, hardware must stop issuing any new transactions. It must record the state of the MFC command so that it can be restarted in the future. However, hardware can complete any current outstanding transactions. If an MFC command is being divided into a series of smaller transactions, hardware must stop the process. Hardware is allowed to complete any transactions divided into a series of smaller transactions before the suspend bit was set. All MFC commands, including any partial transactions, must be flushed before setting the suspended status. Hardware must purge all commands (partial or complete) once the purge bit is set, but it can complete any current outstanding transactions.

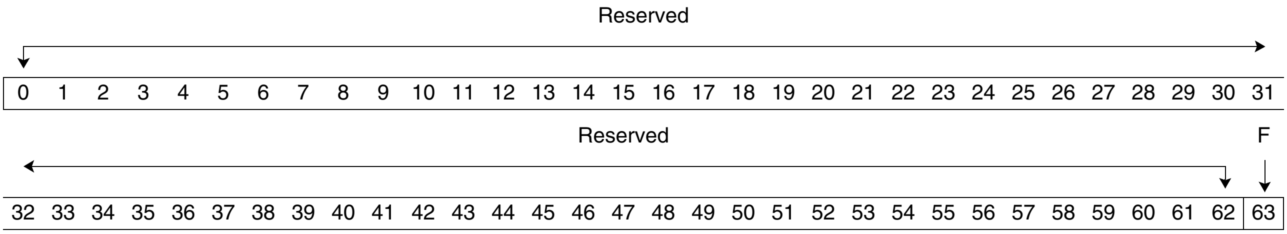
15.11 MFC Atomic Flush Register(MFC_Atomic_Flush)

The MFC Atomic Flush Register is implementation-dependent and access is privileged. Privileged software uses this register to clear the contents of the cache used for atomic DMA commands and releases any current reservations. Data in the cache that is considered modified is pushed to memory, and the line is invalidated. Valid lines in the cache that are not considered modified are invalidated.

For this operation to work properly, privileged software must suspend the MFC command queues.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0200’; where n is the SPE number



Bits	Field Name	Description
0:62	Reserved	Reserved.
63	F	Flush enable or status. When this bit is set to a ‘1’ by software, the contents of the atomic unit are flushed. Hardware resets the bit when the flush operation is complete. Software should never write a ‘0’ to this field. Software should poll this register for completion of the flush operation.

15.12 SPU Outbound Interrupt Mailbox Register (SPU_OutIntrMbox)

The SPU Outbound Interrupt Mailbox Register is used to read 32 bits of data of SPU outbound interrupt mailbox queue of the corresponding SPU. The SPU Outbound Interrupt Mailbox Register has a corresponding channel for writing data into the SPU Write Outbound Interrupt Mailbox Channel (see page 123).

A write channel (**wrch**) instruction sent to the SPU outbound interrupt mailbox queue loads the 32 bits of data specified in the instruction into a mailbox queue for other processors or devices to read.

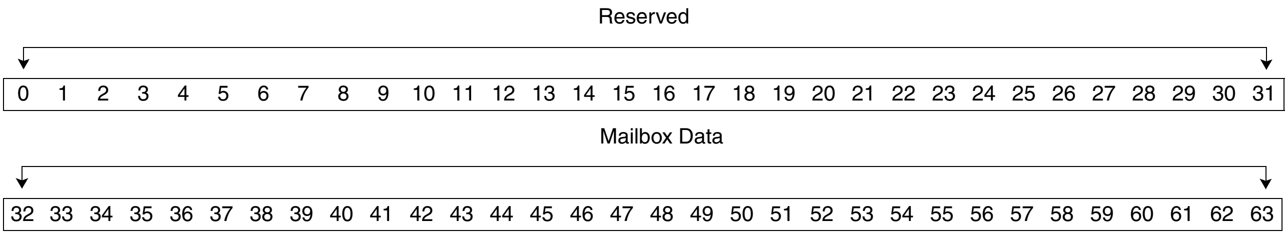
An MMIO read from this register always returns the data in the order written by the SPU. The data returned on a read of the SPU Outbound Interrupt Mailbox queue is undefined.

An MMIO read of the SPU Mailbox Status Register (see page 93) returns the status of the mailbox queues. The number of valid queue entries in the SPU Outbound Interrupt Mailbox Register is given in the SPU_Out_Intr_Mbox_ Count field of the SPU Mailbox Status Register (that is, SPU_Mbox_Stat [SPU_Out_Intr_Mbox_ Count]).

An MMIO read of the SPU Outbound Interrupt Mailbox Register sets a pending SPU Outbound Interrupt Mailbox Available Event. If the amount of data remaining in the mailbox queue is below an implementation-dependent threshold, and if this condition is enabled (that is, SPU_EventMask[Me] set to '1'), the SPU Read Event Status Channel is updated (that is, SPU_EventStat[Me] is set to '1') and its channel count is set to '1'. This causes an SPU Outbound Interrupt Mailbox Available Event (see page 150).

Access Type MMIO: Read

Base Address Offset (BP_Base | P2(n)) + x'04000'; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Mailbox Data	Application-specific mailbox data. Each application can uniquely define the mailbox data.

Implementation Note:

The MFC must not acknowledge the write of the SPU Outbound Interrupt Mailbox Register until a processor or other device has read the contents of the mailbox.

16. SPU Privileged Facilities

SPU Privileged Facilities include these registers:

- SPU Privileged Control Register
- SPU Local Storage Limit Register (see page 217)
- SPU Channel Access Facility, which includes:
  - SPU Channel Index Register (see page 218)
  - SPU Channel Data Register (see page 219)
  - SPU Channel Count Register (see page 220)
- SPU Configuration Register (see page 221).

16.1 SPU Privileged Control Register (SPU_PrivCntl)

The SPU Privileged Control Register provides privileged software with the ability to control the execution environment of the SPU. The SPU Privileged Control Register can be used to place the SPU into single-instruction-step mode or to generate an Privileged Attention Event.

Single-instruction-step mode remains in effect until cancelled by writing this register with the single-step-mode bit reset. When single-step mode is active and the SPU is started using the SPU Run Control Register (see page 86) or the SPU start command on a DMA operation, a single instruction or instruction group is executed. The SPU is stopped, and a Class 2 SPU_Trapped interrupt (if enabled) is presented to the PowerPC Processor Element (PPE). The stopped-by-single-step indicator is set in the SPU Status Register. Other stop conditions can also be reported along with a single-step stop. Single-step operation is not available when the SPU is operating in isolate mode (the isolate [IS] bit in the SPU Status Register = ‘0’). Setting the single-step mode is ignored if the SPU is in an isolated state.

When this register is written with the attention event request bit set, an Privileged Attention Event is raised on the SPU. This condition is reset by SPU acknowledgment of this event. When reading from the SPU Channel Count Register, the current state of single-step mode is provided. However, the attention event request bit always returns ‘0’.

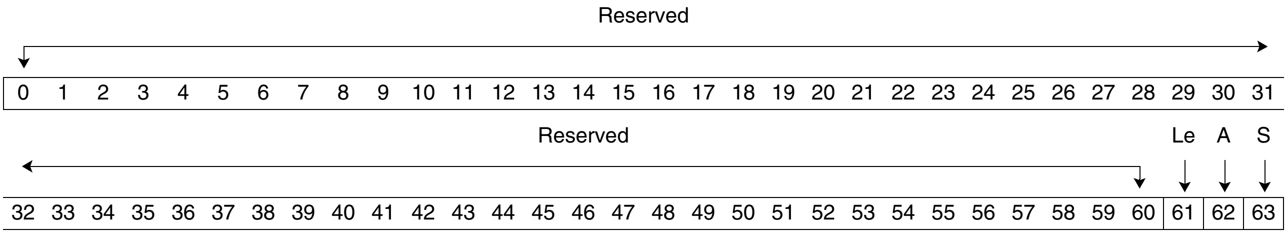
Privileged code can use the Privileged Attention Event mechanism to trigger an SPU event when the SPU software supports the SPU Privileged Attention Event (the Privileged Attention Event is enabled). When the SPU software supports the SPU Privileged Attention Event, it can support requests specific to the operating environment such as light-weight, application-assisted context switching of SPUs.

Privileged code can use the load enable to prevent an application from issuing an isolation load request to put the SPU into an isolated state.

Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04040’; where n is the SPE number



Bits	Field Name	Description
0:60	Reserved	Set to zeros.
61	Le	Isolation load request enable. 0 PPE or SPU isolate load request ignored. 1 PPE or SPU isolate load request allowed. Writing ‘11’ to the SPU Run Control Register (see page 86) causes the SPU to transition into the isolated load state.
62	A	Attention event required. 0 No SPU Privileged Attention Event requested. 1 SPU Privileged Attention Event requested.
63	S	Single-step mode. 0 Normal operation. 1 SPU will issue an instruction or set of instructions and then stop.

Programming Note:

The PPE privileged code can use the Privileged Attention Event (see page 154) to trigger an SPU event when the Privileged Attention Event is supported by SPU software (that is, the Privileged Attention Event is enabled). Software can use this feature to support operating environment specific requests such as light-weight, application-assisted context switching of SPUs.

16.2 SPU Local Storage Limit Register (SPU_LSLR)

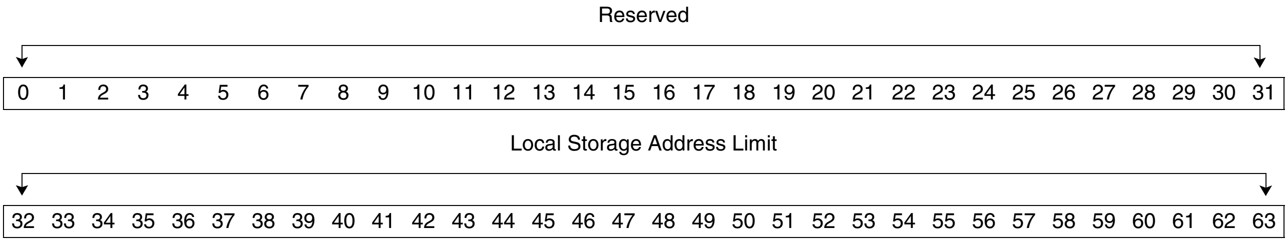
The SPU Local Storage Limit Register allows the size of local storage available to an application to be artificially limited. This register enables privileged software to provide backwards compatibility for applications that are sensitive to the size of local storage. If an application performs a quadword load or store from the SPU that is beyond the range of the SPU Local Storage Limit Register, the operation occurs at the wrapped address.

When an isolation load is requested, the contents of the SPU_LSLR are forced to the maximum value for the implementation. The contents of the SPU_LSLR are not restored to the previous value when exiting an isolated state.

Access to this register is privileged.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x'04058'; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Local Storage Address Limit	Implementation dependent.

16.3 SPU Channel Access Facility

The SPU Channel Access Facility initializes, saves, and restores the SPU channels. The facility consists of three MMIO registers: the SPU Channel Index Register, the SPU Channel Count Register, and the SPU Channel Data Register. The SPU Channel Index Register is a pointer to the channel whose count and data is accessed by the SPU Channel Count Register and SPU Channel Data Register, respectively.

Note: There is also an internal Pending Event register that is described in the *Section 9.11 SPU Event Facility* beginning on page 133.

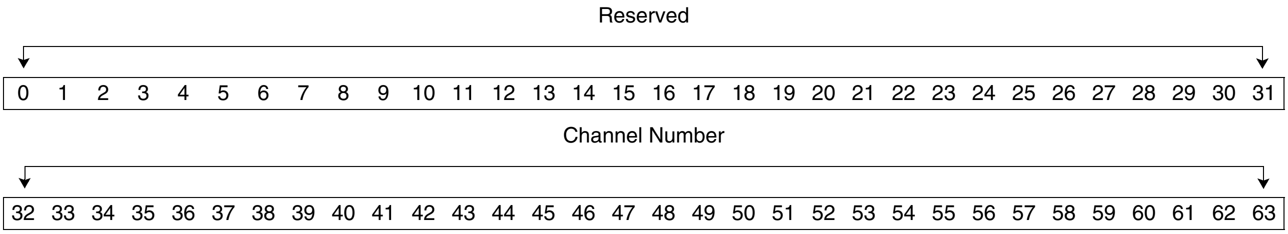
16.3.1 SPU Channel Index Register (SPU_ChnlIndex)

The SPU Channel Index Register selects which SPU channel is accessed using the SPU Channel Count Register or the SPU Channel Data Register.

Access to this register is privileged.

Access Type Write[1]

Base Address Offset (BP_Base | P2(n)) + x‘04060’; where n is the SPE number



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Number	This field contains the channel number that is modified by the SPU Channel Count Register or the SPU Channel Data Register. The number of bits implemented for this field is implementation dependent. Refer to the specific implementation documentation for more information.

[1] Read should be supported for diagnostic purposes.

16.3.2 SPU Channel Data Register (SPU_ChnlData)

The SPU Channel Data Register is used to read or to initialize the SPU channel data selected by the SPU Channel Index Register (see page 218). Initializing or restoring channel data with this register has no effect on the channel count associated with the channel, nor does it generate any channel packet activity on the channel interface.

When the channel being serviced by the channel data and index ports supports more than one-deep FIFOs, writing the channel index to specify the channel will select the oldest FIFO entry to be accessed by the channel data port. Successive accesses to the channel data port will then access successively newer entries in the FIFO. The behavior of accesses beyond the depth of the FIFO is implementation dependent and should be avoided.

Reading or writing the SPU Event Status data through this interface provides direct access to the internal pending event register. Therefore, the external event mask has no effect on this data when reading the channel using a read channel (**rdch**) instruction.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04070’; where n is the SPE number

Reserved

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Channel Data

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Data	This field is used to initialize the data for the channel identified by the SPU Channel Index Register. Access to this resource is privileged. The number of bits implemented for this register is channel specific. Only a subset of the implemented channels can be accessed through this register. Refer to the specific implementation documentation for more information.

Programming Note:

To support SPE context save, restore, and debug, the following channels must be supported through this interface:

- SPU Pending Event Register (supporting SPU Read Event Status Channel x’0’)
- SPU Write Event Mask Channel (x‘1’)
- SPU Signal Notification 1 Channel (x‘3’)
- SPU Signal Notification 2 Channel (x‘4’)
- MFC Read Tag-Group Status Channel (x‘18’)
- MFC Read List Stall-and-Notify Tag Status Channel (x‘19’)
- MFC Read Atomic Command Status Channel (x‘1B’)
- SPU Read Inbound Mailbox Channel (x‘1D)

Channel data for channels x‘0’, x‘1’, x‘3’, x‘4’, x‘18’, x‘19’, x‘1B’, and x‘1D’ must be initialized to zero by privileged software before a new context is started in the SPE.

16.3.3 SPU Channel Count Register (SPU_ChnlCnt)

Each channel has a data port and an associated depth or count (that is, the number of entries in the channel). The channel count value is a record of the number of entries in the channel. The SPU Channel Count Register is used to read or initialize the count associated with the channel selected by the SPU Channel Index Register.

Access to this register is privileged.

Channels are also defined as blocking or non-blocking. Blocking channels will stall the execution of the SPU if the channel is full (on writes) or empty (on reads); that is, the SPU will stall with a channel count of zero.

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04068’; where n is the SPE number

Reserved

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Channel Count

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Channel Count	This field is used to modify the count parameter for the channel identified by the SPU Channel Index Register.The number of bits implemented for this field is channel specific (see the channel descriptions for the maximum value of the count). Refer to the specific implementation documentation for more information.

Implementation Note:

The following channels have counts that must be initialized or restored via this interface:

- SPU Pending Event Register (supporting SPU Read Event Status Channel x’0’)
- SPU Write Event Mask Channel (x‘1’)
- SPU Signal Notification 1 Channel (x‘3’)
- SPU Signal Notification 2 Channel (x‘4’)
- MFC Read Tag-Group Status Channel (x‘18’)
- MFC Read List Stall-and-Notify Tag Status Channel (x‘19’)
- MFC Read Atomic Command Status Channel (x‘1B’)
- SPU Read Inbound Mailbox Channel (x‘1D)

Note: Channel counts for channels x‘0’, x‘3’, x‘4’, x‘18’, x‘19’, x‘1B’, and x‘1D’ must be initialized to zero. Channel counts for channels x‘17’, x‘1C’, and x‘1E’ must be initialized to one. The channel count for theMFC Command Opcode Channel (see page 103) must be initialized to ‘16’ by privileged software before a new context is started.

16.4 SPU Configuration Register (SPU_Cfg)

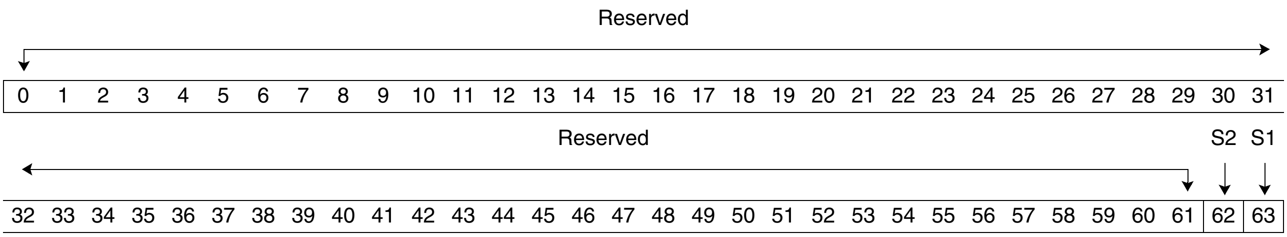
The SPU Configuration Register is used to read or set the configuration of the SPU signal notification registers:SPU Signal Notification 1 Register (see page 94) and SPU Signal Notification 2 Register (see page 95) in the SPUs.

Each SPU signal notification register can be configured to either overwrite the current contents of the register when written or to logically OR the data written with the current contents. The current contents are reset to zero when read using an SPU channel read (**rdch**) instruction.

The mode of each SPU signal notification register must be initialized to overwrite at power-on reset (POR).

Access Type Read/Write

Base Address Offset (BP_Base | P2(n)) + x‘04078’; where n is the SPE number



Bits	Field Name	Description
0:61	Reserved	Reserved.
62	S2	Signal notification 2 mode. 0 Signal notification 2 mode is overwrite (POR default). 1 Signal notification 2 mode is logical OR.
63	S1	Signal notification 1 mode. 0 Signal notification 1 mode is overwrite (POR default). 1 Signal notification 1 mode is logical OR.

17. SPE Context Save and Restore

Saving and restoring the context of an SPU element (SPE) unit can be very expensive in terms of time and system resources. An SPE allocation scheme following a “serially reuseable device” model, in which an SPE is assigned a task until it completes, then another task is assigned in a serial fashion, results in better utilization of SPEs. The “serially reuseable device” model required less context to be saved and restored during a context switch.

When this is not possible, the hardware supports the capability to suspend a task on the SPE, fully save its context, fully restore that context at a later time, and resume the task. Though facilitated by hardware, context save and restore is software intensive. A preemptive context switch facilitated by privileged software is the most costly form of context save and restore, since the context is not known and a worst case save and restore of all contexts must be performed. The use of application yielding, in which the application using the SPU determines the timing and the amount of context saved and restored can provide significant cost savings in terms of cycles and space used to save and restore the SPE context. However, the application primarily drives the technique selected, and the architecture supports the full context save and restore of the SPE. The Context Save and Context Restore sequences in the following sections handle the cases where the SPU is already stopped at Context Save time. Some operating environments can forgo restoring the context of an SPE that has been stopped (especially when stopped on error conditions). However this support is included here for completeness.

Note: Preemptive context switching of an SPU that interfaces directly with an I/O device should be specifically avoided since the physically mapped local storage is considered part of the context.

The context save and restore sequences are described in the specific implementation documentation.

18. PPE Address Range Facility

The PowerPC Processor Element (PPE) address range facility provides a method to map effective addresses or real addresses (in real mode) for PPE loads and stores and instruction fetches to a class ID for the cache replacement management facility (see *Section 19.1 Replacement Management Table* beginning on page 231). A set of PPE address range registers is provided to implement this facility.

An implementation should provide a minimum of two sets of range registers for instruction fetches and two sets of range registers for data fetches per logical PPE (that is, a thread in the processor where MSR[IR] = '0' and MSR[DR] = '0'). PPE Special Purpose Registers (SPRs) are provided for these four range registers. The SPRs should be duplicated for each processor thread.

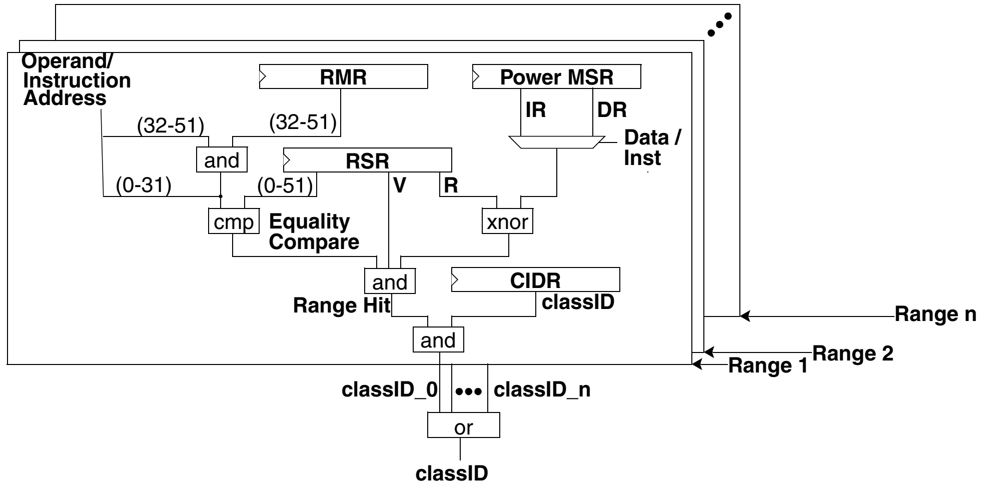
The PPE address range facility consists of:

- Range Start Register (see page 227)
- Range Mask Register (see page 228)
- Class ID Register (see page 229)

An address range is a naturally-aligned range that is a power of 2 in size and is between 4 KB and 4 GB, inclusive. An address range is defined by two registers; the Range Start Register (RSR) and the Range Mask Register (RMR). For each address range, there is an associated ClassID Register (CIDR) that specifies the class ID. These address range registers are accessible by the PPE move-to or move-from special-purpose register instructions. Access to these registers is only allowed in privileged state.

Figure 18-1 illustrates how the address range registers are used to generate the class ID.

Figure 18-1. Generation of Class ID from the Address Range Registers



If all the following conditions are met, the particular address range defined by an RSR and RMR pair applies to a given effective address, and a “range hit” is said to have occurred. Then the Class ID from in the corresponding Class ID register is used as an index into the RMT. For example:

- RSR[63] = ‘1’
- RSR[0:51] = EA[0:31] || (EA[32:51] & RMR[32:51])
- If the operation is a load or store, then
    - RSR[62] = MSR[DR]
- else (the operation is an instruction fetch)
    - RSR[62] = MSR[IR]

If there is no “range hit” for a given effective address, the class ID has a value of zero. In effect, the RMR defines the size of the range by selecting the bits of an effective address used to compare with the RSR. The upper bits of an RSR contain the starting address of the range and the lower bits contain a relocation mode (virtual or real) and a valid bit. The size of the range must be a power of two. The starting address of the range must be a range size boundary.

Note: All cache management instructions should be treated as loads or stores for calculating the ClassID, or the operation.

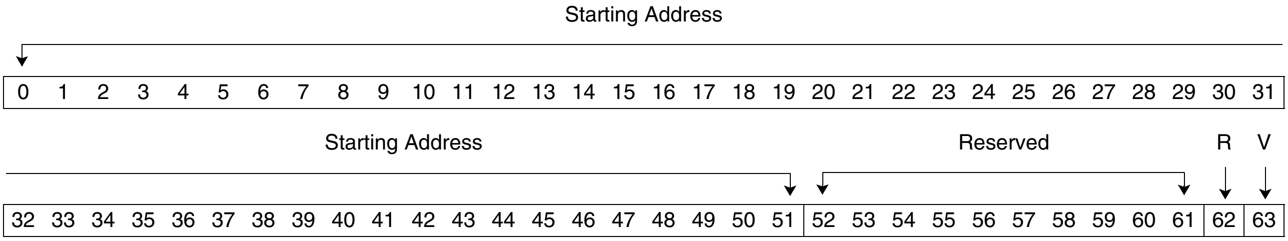
Programming Note:
To avoid confusion about the classID value, software should ensure that the address ranges specified in the PPE Address Range facility do not overlap (that is, more than one range has a simultaneous hit). If two address ranges are hit by the same address, the resulting classID will be a logical OR of the two values in the Class ID Register (CIDR).

18.1 Range Start Register (RSR)

The Range Start Register contains the starting address of the PPE address range. The upper bits of this register contain the starting address; the lower bits contain an address range mode (real or virtual). The size of the range must be a power of two, and the starting address must be a range-size boundary.

Access Type Read/Write

SPR Offset Implementation dependent



Bits	Field Name	Description
0:51	Starting Address	This field contains the upper 52-bits of the starting address for the range.
52:61	Reserved	Set to ‘0’.
62	R	Relocation mode. Used to compare the instruction relocate (IR) or data relocate (DR) bit of the Power Machine State Register (MSR). 0 The address relocation via effective-to-virtual-address translation is off (real addressing mode). 1 The address relocation via effective-to-virtual-address translation is on (virtual addressing mode).
63	V	Range register valid. 0 Range register disabled. 1 Range enabled.

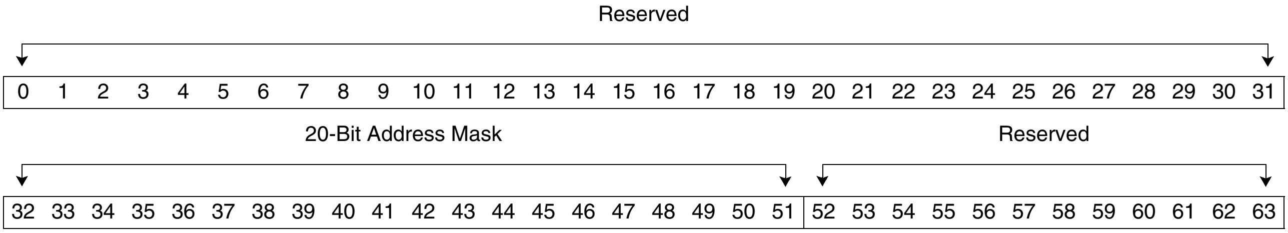
18.2 Range Mask Register (RMR)

The Range Mask Register defines the size of the PPE address range by selecting the bits of an operand address used to compare with the Range Start Register.

Bits 32:51 of the operand or instruction address is ANDed with the 20-bit RMR. Bits 0-31 of the operand or instruction address is then concatenated with the result of the AND and compared with the starting address in the RSR to determine a range hit. The upper 32 bits are always compared.

Access Type Read/Write

SPR Offset Implementation dependent



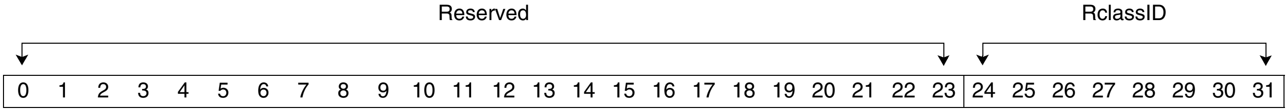
Bits	Field Name	Description
0:31	Reserved	Set to '1'. This reserve field is set differently from others as explained in *Section 1.4 Reserved Fields and Registers* beginning on page 25.
32:51	20-Bit Address Mask	Corresponds to effective address bits 32 through 51.
52:63	Reserved	Set to zeros.

18.3 Class ID Register (CIDR)

The Class ID Register contains the RclassID to use when the address of the operand or instruction matches the PPE address range.

Access Type Read/Write

SPR Offset Implementation dependent



Bits	Field Name	Description
0:23	Reserved	Reserved.
24:31	RclassID	The size of this field is implementation dependent. Refer to the specific implementation documentation for more information.

19. Cache Replacement Management Facility

The Cell Broadband Engine Architecture (CBEA) provides a method of controlling the cache replacement based on a replacement class identifier (RClassID). The class ID is provided as a parameter in the MFC instructions and is generated from the load-and-store address for PowerPC Processor Element (PPE) operations (see *Section 18* on page 225). The class ID is used to generate an index to a privileged-software managed table, which is used to control the replacement policy.

The format of the Replacement Management Table (RMT) is implementation-dependent. Any CBEA-compliant implementation should provide an RMT for each major cache structure. An example of an RMT and the index generation method is provided in Replacement Management Table and for the RMT Index Generation Example (see page 232).

In this version of the CBEA, MMIO register locations are provided for an L2 Replacement Management Table (RMT) and an MFC TLB RMT. PPE special purpose registers are provided for a PPE TLB RMT.

See the specific implementation documentation for details on the support of the Cache Replacement Management facility.

19.1 Replacement Management Table (RMT) Example

The cache replacement is controlled by privileged software through an RM. Each level of cache including the TLBs that supports replacement management must have an independent RMT.

The RMT consists of an implementation-dependent number of entries, which should contain set-enable bits, a valid bit, and other control information. Optionally, an implementation can also provide a cache bypass bit and an algorithm bit. The number of entries and the size of each entry in the RMT table is implementation-dependent. *Table 19-1* depicts a typical RMT entry for an 8-way, set-associative cache. The RMT table is located in the real address space of the system. The privileged software should map these RMT tables as privileged pages. An implementation should provide an RMT for each major cache structure.

Table 19-1. Typical RMT Entry for an 8-Way Set Associative Cache

S0	S1	S2	S3	S4	S5	S6	S7	Reserved	a	b	v
0	1	2	3	4	5	6	7	8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28	29	30	31

Bits	Field Name	Description
0:7	S<n>	Cache set enable: n = 0 — n = 7.
8:28	Reserved	
29	a	The algorithm bit specifies the replacement algorithm to be used for this class. 0 Least recently used (LRU) 1 Most recently used (MRU)
30	b	The bypass bit indicates that the operation should not be cached at this level. (Not valid for translation RMTs.)
31	v	The valid bit indicates that the RMT entry contains valid information.

The RMT defines which sets in the set-associative cache are to be used for the respective replacement management class. If the set-enable bit for the respective set is ‘1’, that set is used by that replacement class in the RMT entry. If the set-enable bit is not set, the associated set is not used for operations involving the respective replacement management class. One or more sets can be used for more than one replacement management class. Using replacement management classes for streaming areas to prevent thrashing[1] the cache is an example of how the cache replacement management facility can be used.

Accessing an invalid RMT results in the default class of x‘0’ being used for the operation. If the default class is also invalid, the class is treated as though all set enables were set.

Setting the bypass bit indicates that the data should not be cached at a hierarchy level that corresponds to this RMT. For data caches, the load or store should bypass the cache and be passed directly to the bus. It is possible that the data for the operation already exists at this hierarchy level. In this case, the implementation can source the data from the cache, provide the data through intervention, or cause the data to be invalidated or pushed from this cache level. In any case, the load or store still follows the normal rules for coherency. A table is generated to show all possible conditions for WIMG[2] settings and cache states.

19.2 RMT Index Generation Example

The replacement-management table (RMT) is indexed for two purposes: to update the contents and to access the contents for management of DMAs and loads and stores.

To update the contents, software sets the RMT Index Register to point to the entry and stores the new data for the RMT entry to the RMT data registers. The RMT Index Register and RMT data registers are implementation-dependent. The index registers are described in *Section 19.2.1 RMT Index Register* on page 233 and an example of the data registers is provided in *Table 19-1 Typical RMT Entry for an 8-Way Set Associative Cache* on page 231. Refer to the specific implementation documentation for more information.

Accessing the RMT for management purposes is system-dependent. Few requirements are placed on the index by the CBEA. Devices that share a cache hierarchy must be able to share the entries in the RMT (the RMT index is the same for two or more devices), or must have an independent area of the table (the RMT index is unique).

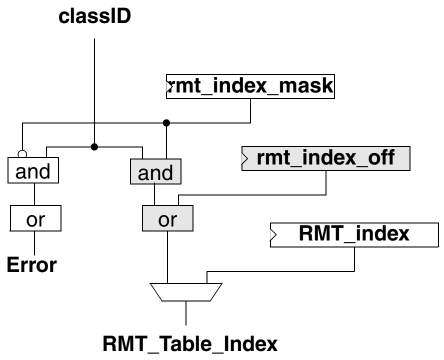
When the cache-hierarchy level is dedicated to a single device, the RMT index can be as simple as a range check on the class ID. In the case of a shared cache, the class ID must be converted to an RMT index. *Figure 19-1 RMT Index Generation* on page 233 illustrates one method of generating the RMT from the class ID. In this example, a pair of registers is used to map the class ID to an RMT index. The RMT Index Mask Register is used to mask off the upper bits of the class ID. The bits disabled by the RMT Index Mask Register are replaced by the RMT Index Off Register. Each device that shares this cache-hierarchy level must have an independent set of mask and offset registers. The shaded items are only required if the RMT is shared by more than one device.

Access to the RMT index mask and RMT index off registers is privileged. Privileged software must set the RMT Index Mask Register with zeros in the upper bits and ones in the lower bits. The number of zeros in the lower bits of the RMT Index Off Register must be equal to or greater than the number of ones in the RMT Index Mask Register. Refer to the specific implementation documentation for more information.

[1] A cache is said to thrash when its miss rate is too high and it spends most of its time servicing misses.

[2] 4 bits in the page table, also called a page table entry, which control processor accesses to cache and to main storage. “W” stands for write through, “I” for caching inhibited, “M” for memory coherence, and “G” for guarded storage.

Figure 19-1. RMT Index Generation



19.2.1 RMT Index Register (RMT_Index)

This register is used to hold the index of the RMT entry that is to be modified using the RMT Data Registers. The RMT Index Register is an optional facility. An RMT Index Register must be provided for each facility that supports the cache replacement policy.

Appendix A — Memory Maps on page 269 lists the base address for the RMT Index Register for the PPE L2 cache, the SPE TLBs, and the SPR offsets for the PPE TLB. Refer to the specific implementation documentation for more information on the implementation of the RMT facility.

Access Type Read/Write

SPR Offset Implementation dependent

Base Address Offset Implementation dependent

Implementation-Dependent

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Index

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63

Bits	Field Name	Description
0:31	Implementation-Dependent	
32:63	Index	The number of bits in this field is implementation-dependent.

19.2.2 RMT Data Register (RMT_Data)

This register is used to access the entry in the RMT pointed to by the RMT Index Register. The RMT Data Register is an optional facility. An RMT Data Register must be provided for each facility that supports the cache replacement policy.

Appendix A — Memory Maps on page 269 lists the Base Address for RMT Data Register for a PPE L2 cache, the SPE TLBs, and the SPR offsets for the PPE TLB.

Refer to the specific implementation documentation for more information on the RMT facility.

Access Type	Read/Write
SPR Offset	Implementation dependent
Base Address Offset	Implementation dependent

Implementation-Dependent

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Implementation-Dependent

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:63	Implementation-Dependent	

20. Resource Allocation Management

Resource allocation management (RAM) provides a mechanism to allocate portions of a resource’s time to a specific resource allocation group (RAG). The number of RAGs is implementation-dependent. The resources that are managed are memory units and I/O interfaces (IOIFs). The definition of a memory unit is implementation-dependent, and could, for example, be defined as a range of addresses or interleaved addresses. The resource allocation groups are groups of one or more hardware units called requesters. These requesters are physical or virtual units; they can initiate load or store requests, or DMA read or write accesses. The requesters are:

- PPE groups (PPE 0, PPE 1, MMU, L2 cache)
- SPEs
- Virtual channels associated with the physical IOIFs

The PPE groups and the SPEs are each in a specific RAG at an instant in time. However, they can be assigned to various RAGs over time. Each RAG can be identified by a resource allocation ID (RAID). Software can configure the RAID for the PPE group or SPE with a memory-mapped register.

Each IOIF has multiple virtual channels—one for each of the resource allocation groups. The external bridge or I/O device must identify which virtual channel is associated with each physical transfer on the IOIF. The I/O controller (IOC) acts as the agent on behalf of the virtual channels. How an I/O device is assigned to a RAG, and whether it can be dynamically changed, is outside the scope of the Broadband Processor and is the responsibility of the external bridge or I/O device.

Refer to the specific implementation documentation for more information.

21. Interrupt Facilities

The Cell Broadband Engine Architecture (CBEA) provides facilities for:

- Routing interrupts and interrupt status information to a PowerPC Processor Element (PPE) or external devices
- Prioritizing interrupts presented to a PPE
- Generating an interprocessor interrupt (IPI)

In the CBEA, interrupts are a result of SPU application actions, errors, or unusual conditions that arise in the execution of instructions or commands. Interrupts directed to a PPE allow the PPE to change state as a result of the interrupt.

In addition, the CBEA defines additional interrupting conditions for the SPEs. Each SPE has a set of interrupt registers for masking the interrupting condition, holding the status of the interrupting conditions, and routing the interrupt to a PPE or other device in the system.

PowerPC Architecture, Book III describes the interrupt definitions, interrupt ordering, interrupt synchronization, and interrupt processing provided under the PowerPC Architecture. The PowerPC Architecture defines a condition that can cause an interrupt as an exception. An interrupt is defined as the change in processor state caused by handling an exception. The CBEA has a different definition of an interrupt.

When enabled, an SPE interrupt condition causes an interrupt to be routed to a PPE or other device which, depending on the state of the PPE, can cause a change in the processor's state. Interrupts routed to a PPE are presented as an “external interrupt exception.”

Note: By masking the interrupting condition, privileged software can also support polling.

21.1 Interrupt Classes

In the PowerPC Architecture, interrupts are classified by cause. “Instruction caused” interrupts are directly caused by the execution of an instruction. “System caused” interrupts are caused by some other system exception. The additional interrupt conditions provided by the CBEA are classified as system caused. They are presented to the PPE as external interrupts. External interrupts are always imprecise with respect to instruction processing in the PPE, and they cause an asynchronous change in state. See *PowerPC Architecture, Book III* for a description of external interrupts.

External interrupts are maskable in the PPE, and a mask for each additional interrupt condition is supported in the CBEA. Refer to *Section 21.6 MFC Interrupt Mask Registers* beginning on page 253 for a description of external interrupt masks. An implementation can also provide other implementation-dependent interrupt status and masks. Refer to the specific implementation documentation for more information.

21.2 Interrupt Presentation

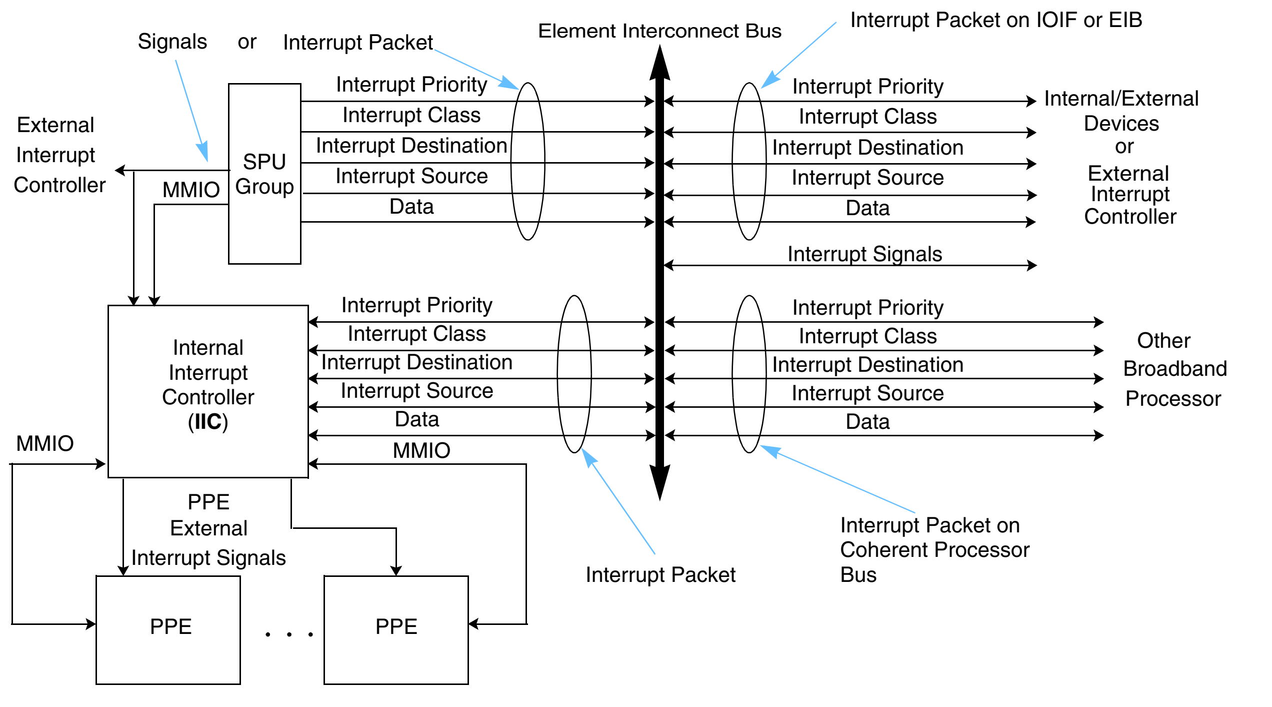
The PPEs in a CBEA-compliant processor are capable of servicing and generating interrupts and will never handle interrupts generated by other processors or devices. SPUs are only capable of generating interrupts. Interrupts generated by an SPU are routed to a PPE or to other devices for processing. Software can generate an interrupt to a PPE.

Figure 21-1 on page 238 illustrates how the interrupts are presented inside a CBEA-compliant processor.

Interrupts generated by an SPU group are sent to an external interrupt controller or to an internal interrupt controller (IIC) using either dedicated signals or an interrupt packet on the internal Element Interconnect Bus (EIB). External devices send interrupts to a CBEA-compliant processor using either dedicated signals or an interrupt packet on the I/O interface (IOIF). An IIC receives the interrupt packet and signals an external interrupt to the appropriate PPE. Software running on a PPE or SPU can cause an interrupt to be sent to a logical PPE by writing the corresponding Interrupt Generation Port Register. As a result of the MMIO write, the IIC signals an interrupt to a PPE.

The IIC interrupt generation, routing, and presentation are described in *Section 21.3 Internal Interrupt Controller Registers* beginning on page 239. SPU group registers related to external interrupt generation, routing, and presentation are described in *Section 21.4 SPU and MFC External Interrupt Definitions* beginning on page 245 through *Section 21.8 Interrupt Routing Register* beginning on page 260.

Figure 21-1. Interrupt Presentation



21.3 Internal Interrupt Controller Registers

The IIC has an interrupt control block for each physical and logical PPE (that is, a thread in the physical processor). These control blocks are mapped in the real address space, starting at an implementation-dependent offset from the BP_Base. The control block's starting address is defined as the BP_Base | IIC (p) + x’400’ + (t * x’20’); where 'p' is the physical PPE and 't' is the thread number for the corresponding PPE. (See *Table A-1 Broadband Processor Memory Map* on page 269 for more details.)

Table 21-1 shows the registers associated with each control block and their offsets from starting address of the BP_Base.

Table 21-1. Internal Interrupt Controller Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
PPU (Thread t) Interrupt Control Block, where 0 ≤ t < number of PPU threads -1			
x‘000’	INT_Pending_NonD	Interrupt Pending Port Registers (see page 239), non-destructive read. Status and data for pending interrupt.	Read Only
x‘008’	INT_Pending_D	Interrupt Pending Port Registers (see page 239), destructive read. Status and data for pending interrupt.	Read Only
x‘010’	INT_Generation	Interrupt Generation Port Register (see page 243). Port for generation of an interprocessor interrupt (IPI).	Write Only
x‘018’	INT_CPL	Interrupt Current Priority Level Register (see page 244). Only higher-priority interrupts cause an external interrupt.	Read/Write

21.3.1 Interrupt Pending Port Registers (INT_Pending_NonD and INT_Pending_D)

The Interrupt Pending Port Registers allow software to read the interrupt packet data and other information about the highest priority interrupt pending for each PPE. There is one Interrupt Pending Port Register for each PPE in a CBEA-compliant processor. The following description applies to interrupts for one PPE and that PPE Interrupt Pending Port Register, Interrupt Current Priority Level Register, and Interrupt Generation Port Register.

When an interrupt packet arrives at the IIC, if an interrupt of the same priority is not already queued in the IIC for the PPE, the interrupt packet is retained in an interrupt pending queue. It is implementation-dependent whether more than one interrupt packet of the same priority is retained or whether subsequent interrupt packets are retried. However, the IIC implementation must always be able to accept at least one interrupt per priority value.

When the priority of the highest priority, valid interrupt in the interrupt pending queue is higher (lower numeric value) than the priority in the current priority level, the external interrupt signal to the PPE is activated. When the priority of the highest priority, valid interrupt in the interrupt pending queue is the same or lower (equal or higher numeric value) than the priority in the Interrupt Current Priority Level Register, the external interrupt signal to the PPE is deactivated.

Software should read the Interrupt Pending Port Register in the first level interrupt handler (FLIH) through an MMIO load to obtain information about the interrupting condition. When software reads the Interrupt Pending Port Register, the IIC returns the value of the highest priority, valid interrupt in the interrupt pending queue. Software can read the Interrupt Pending Port Register with either of two addresses. One address is used for destructive reads, and the other address is used for non-destructive reads. When the Interrupt Pending Port Register is read destructively, the interrupt valid bit for the highest priority, valid interrupt in the interrupt pending queue is set to ‘0’. The priority value of this interrupt is copied into the Interrupt Current Priority Level

Register. Thus, a destructive read causes the external interrupt signal to the PPE to be deactivated since the Interrupt Current Priority Level Register value represents the highest priority of the pending interrupts. This external interrupt signal to the PPE remains deactivated until one of two events occurs. A higher priority interrupt packet arrives at the IIC, or software lowers the priority in the Interrupt Current Priority Level Register and a higher priority interrupt packet is or becomes pending. When the Interrupt Pending Port Register is read non-destructively, the valid bit in the interrupt pending queue is not changed, and the Interrupt Current Priority Level Register is not changed.

When the Interrupt Pending Port Register is read (destructively or non-destructively) and the valid bit is ‘1’, the data returned for the highest priority interrupt includes the following information about the interrupt: valid bit, type, class, interrupt source (ISRC), priority and, optionally, the interrupt packet data. If the interrupt packet data is not supported by an implementation, the interrupt packet data is returned as zeros. If the type bit is ‘0’, the interrupt packet originated from an MFC, external device, or external interrupt controller. If the type bit is ‘1’, the interrupt packet originated from the Interrupt Generation Port Register, and the class information and ISRC are read as zeros.

If software reads the interrupt pending port (destructively or non-destructively) when there is no valid pending interrupt, the valid bit returned will be zero, and the value returned for the other fields is undefined. If software reads the interrupt pending port when there is no valid pending interrupt, the Interrupt Current Priority Level Register is unchanged.

If a non-destructive read of the interrupt pending port that returns a valid interrupt is followed by another read of the interrupt pending port (destructive or non-destructive), the IIC returns the same data value unless another interrupt of a higher priority has been received by the IIC. For example, there are two pending interrupts of the same priority: one due to an external device sending an interrupt packet, and one due to a store to the interrupt generation port (IGP). Once software reads the interrupt pending port non-destructively and gets the interrupt pending port value for one of these interrupts, the same interrupt pending port value is returned on the next interrupt pending port read, assuming no interrupt of a higher priority has been received by the IIC.

When a valid interrupt of a specific priority exists in the interrupt pending queue, other pending interrupts of the same priority and destination can be queued at other points in the CBEA-compliant processor or externally. Reading the Interrupt Pending Port Register destructively resets the interrupt valid bit, allowing a subsequent interrupt to move into the interrupt pending queue. The latency for a pending interrupt to move to the Interrupt Pending Port Register is implementation-dependent.

An implementation can support fewer bits than architected for interrupt packet data, interrupt class, ISRC, and priority. If an implementation supports fewer priority bits than the number architected, the most significant bits of the priority field should be supported in order to get a more consistent view of priority in an environment where implementations or external devices support different numbers of bits. An implementation with N priority bits divides the full set of 256 priority values into 2^N ranges, so that priorities from implementations with all priority bits that fall in the different 2^N ranges still appear in their priority order with respect to each other.

If software synchronizes the MMIO load for a destructive read of the interrupt pending port, and then sets the Machine State Register (MSR) external interrupt enable bit is to ‘1’, no external interrupt will occur at the same or a lower priority until software directly or indirectly modifies the Interrupt Current Priority Level Register.

Implementation Note:

For the deactivation of the external interrupt property to function correctly, hardware must ensure that the data read from the interrupt pending port is always returned before the external interrupt signal is deactivated.

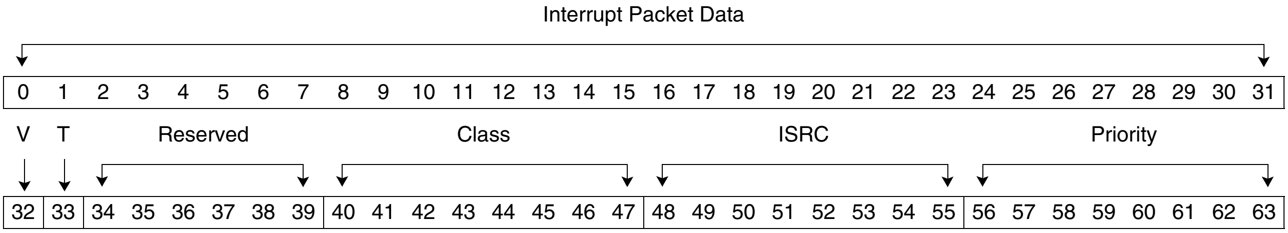
The interrupt pending port acts as a window into the interrupt pending queue. Software always see the highest priority interrupt in this window at the time the interrupt pending port is read. This view of the interrupt pending queue is independent of the Interrupt Current Priority Level Register (see page 244).

After a valid interrupt is pending in the interrupt pending queue, software can cause the external interrupt signal to the PPE to be activated or deactivated by writing the Interrupt Current Priority Level Register. Software can also mask interrupts from the IIC by writing the current priority level value to zero.

A flat interrupt priority scheme, in which all interrupts have the same priority, can be implemented as follows: Set the same priority value for all classes in the Interrupt Routing Register and configure the same priority value for interrupts associated with external devices and any external interrupt controller. After the destructive read of the Interrupt Pending Port Register, which software performs for each occurrence of a PPE external interrupt, software must store a lower priority (higher numeric value) to the Interrupt Current Priority Level Register to enable the presentation of the next interrupt packet.

Access Type Read Only

Base Address Offset BP_Base | IIC(p) + x’400’ + (t * x’20’) + x‘000’; Non-destructive read
BP_Base | IIC(p) + x’400’ + (t * x’20’) + x‘008’; Destructive read
(where p is the PPE number and t is the PPE (p) thread number)



Bits	Field Name	Description
0:31	Interrupt Packet Data	The meaning of the data in this field is dependent on the interrupt source. (Optional)
32	V	Interrupt valid.
33	T	Interrupt type. 0 Interrupt from MFC, external device, or external interrupt controller. 1 Interrupt from interrupt generation port.
34:39	Reserved	Reserved.
40:47	Class	Interrupt class.
48:55	ISRC	Interrupt source.
56:63	Priority	Interrupt priority.

21.3.2 Interrupt Generation Port Register (INT_Generation)

The Interrupt Generation Port Register allows privileged software to generate an interrupt packet to a PPE. There is one Interrupt Generation Port Register for each PPE in the CBEA-compliant processor. Software can generate an interrupt packet to a PPE by storing to the PPE Interrupt Generation Port Register. This interrupt packet does not need to be transmitted on the internal CBEA-compliant processor interconnect bus, since the Interrupt Generation Port Register and the destination of the interrupt packet are both in the same internal interrupt controller (IIC). The least significant byte written to this register contains the interrupt priority. When the interrupt packet is read through the Interrupt Pending Port Register, the interrupt packet data, class information, and ISRC are read as zeros.

If there are multiple stores to a PPE interrupt generation port with the same priority value, and thus multiple pending interprocessor interrupts to a PPE with the same priority, interrupt packets of the same priority can be merged and only a subset of these interrupts presented.

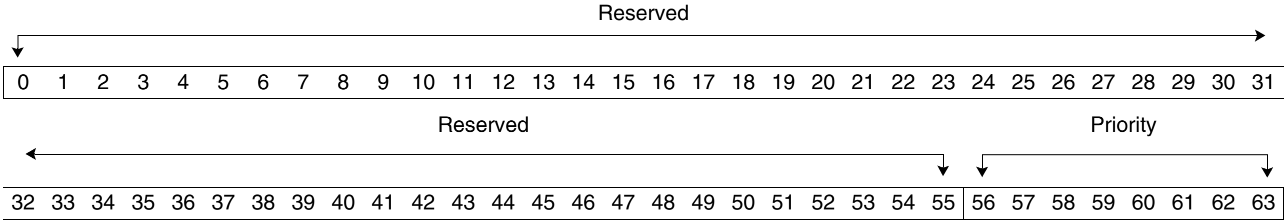
An implementation can support fewer bits than the architecture provides for priority. If an implementation supports fewer priority bits than the number provided, the supported bits must be in the most significant bit positions of the priority field.

Programming Note:

For interprocessor interrupts, software can need to use other means, such as a message queue in memory, to convey information such as class, interrupt source, and the reason for the interrupt. Since a subset of interrupts of the same priority to a PPE might be presented through the interrupt pending port, a message queue in memory could also be used to convey information about individual interrupts with the same priority.

Access Type Write Only

Base Address Offset BP_Base | IIC(p) + x’400’ + (t * x’20’) + x‘010’
(where p is the PPE number and t is the PPE (p) thread number)



Bits	Field Name	Description
0:55	Reserved	
55:63	Priority	Interrupt priority.

21.3.3 Interrupt Current Priority Level Register (INT_CPL)

There is one Interrupt Current Priority Level Register for each PPE in the CBEA-compliant processor. It holds the priority level at which software is currently operating. The lower the numeric value of the priority field, the higher the priority level is. Thus, the highest priority corresponds to a numeric value of zero. The priority level value can be written explicitly by software storing to the Interrupt Current Priority Level Register, or indirectly by software performing a destructive read of the interrupt pending port. See *Section 21.3.1 Interrupt Pending Port Registers* for details of interrupt pending port reads.

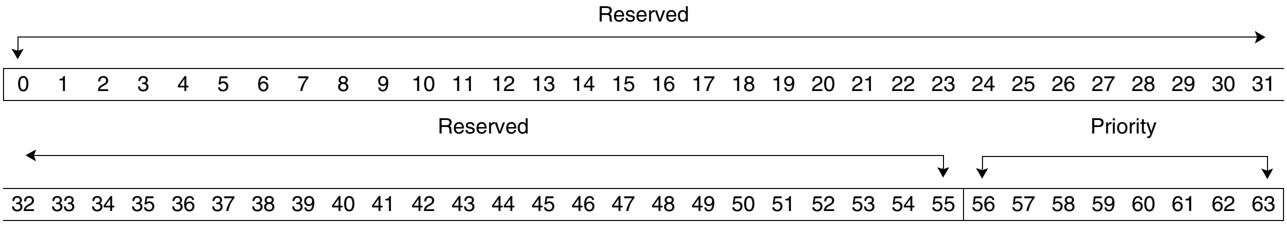
Software can mask interrupts at and below a specific priority level by storing the desired priority value in the Interrupt Current Priority Level Register. To synchronize the interrupt masking side effects of this store, software must read the current priority level and synchronize the MMIO load for this read. After this synchronization, an external interrupt will not occur unless its priority is higher than the priority value in the Interrupt Current Priority Level Register.

Implementation Note:

For the deactivation of the external interrupt property to function correctly, hardware must ensure that the data read from the Interrupt Current Priority Level Register is always returned before the external interrupt signal is deactivated.

Access Type Read/Write

Base Address Offset BP_Base | IIC(p) + x’400’ + (t * x’20’) + x‘018’
(where p is the PPE number and t is the PPE (p) thread number)



Bits	Field Name	Description
0:55	Reserved	Reserved.
56:63	Priority	Current priority level (0 is highest priority).

21.4 SPU and MFC External Interrupt Definitions

Table 21-2 and *Table 21-3* show the additional interrupt definitions provided by the CBEA. These interrupt definitions share the same external interrupt vector defined in *PowerPC Architecture, Book III*. The interrupt definitions listed in *Table 21-2* and *Table 21-3* are generated by the MFC and SPU within a CBEA-compliant processor.

Table 21-2. SPU and MFC Interrupt Class Definitions

Class	Meaning	Description
0	Error	This interrupt class is used for all error conditions relating to SPU group processing and DMA transfers.
1	Translation	This interrupt class is used for all translation exceptions relating to a DMA transfer.
2	Application	This interrupt class is used for all application interrupts.
3	Reserved	This interrupt class is reserved for future use.

Table 21-3. SPU and MFC External Interrupt Definitions

Class	Interrupt Type	Description
0	DMA Alignment Error Interrupt	This interrupt occurs when software attempts to execute a DMA command that is not aligned.
0	Invalid DMA Command Interrupt	This interrupt occurs when software attempts to execute a DMA command with an invalid opcode, an MFC command not supported for the specific queue (for example, MFC atomic commands in the MFC proxy command queue), or an invalid form of an MFC command.
0	SPU Error Interrupt	This interrupt occurs when the SPU has encountered an error condition. These conditions are listed below: • Illegal channel instruction detected. • Invalid instruction detected. • Other implementation-dependent errors. (See the documentation for a specific implementation.) All these conditions should be individually maskable in the SPU.
1	MFC Data Segment Error Interrupt	This interrupt occurs when the DMA effective address cannot be translated to a virtual address.
1	MFC Data Storage Error Interrupt	This interrupt occurs when the DMA effective address cannot be translated to a real address.
1	MFC Local Storage Address Compare Suspend on **get** Interrupt	This interrupt occurs when MFC command queue operation is suspended (for both queues) due to an MFC Local Storage Address Compare match. A match occurs when the local storage address of a MFC DMA command matches the range specified in the MFC Local Storage Address Compare Register when writing to the local storage and the MFC_ACCR[Lg] bit is set to '1'. See *Section 15.6* on page 203 and *Section 15.7* on page 205.
1	MFC Local Storage Address Compare Suspend on **put** Interrupt	This interrupt occurs when MFC command queue operation is suspended (for both queues) due to an MFC Local Storage Address Compare match. A match occurs when the local storage address of a MFC DMA command matches the range specified in the MFC Local Storage Address Compare Register when from the local storage and the MFC_ACCR[LP] bit is set to '1'. See *Section 15.6* on page 203 and *Section 15.7* on page 205
2	Mailbox Interrupt	This interrupt occurs when an SPU writes to an SPU Write Outbound Interrupt Mailbox Channel (see page 123).
2	SPU Stop-and-Signal Instruction Trap	This interrupt occurs when an SPU executes a stop-and-signal (**stop**) instruction.
2	SPU Halt Instruction Trap or SIngle Instruction Step Complete	This interrupt occurs when an SPU executes a halt conditional instruction, and the condition is met.
2	Tag-Group Completion Interrupt	This interrupt occurs when all commands for a selected tag group or tag groups are complete. The interrupt generation is dependent on the Proxy Tag-Group QueryMask Register and the Proxy Tag-Group QueryType Register. See *Section 8.4* beginning on page 82 for more details.
2	SPU Inbound Mailbox Threshold Interrupt	This interrupt occurs when the number of valid entries in the SPU Inbound Mailbox queue is below an implementation-dependent value or threshold. Valid entries are removed from the queue by an SPU application reading from the SPU Inbound Mailbox Channel. See *Section 8.6* beginning on page 90 for more information on the mailbox facility.

21.5 SPU and MFC Interrupt Generation Process

There are three classes of interrupts generated:

- Class 0 Interrupts.
- Class 1 Interrupts (see page 249)
- Class 2 Interrupts (see page 251)

21.5.1 Class 0 Interrupts

As shown in *Figure 21-2* on page 248, there are two SPU exceptions that cause the SE bit in the MFC Class 0 Interrupt Status Register to be set to ‘1’:

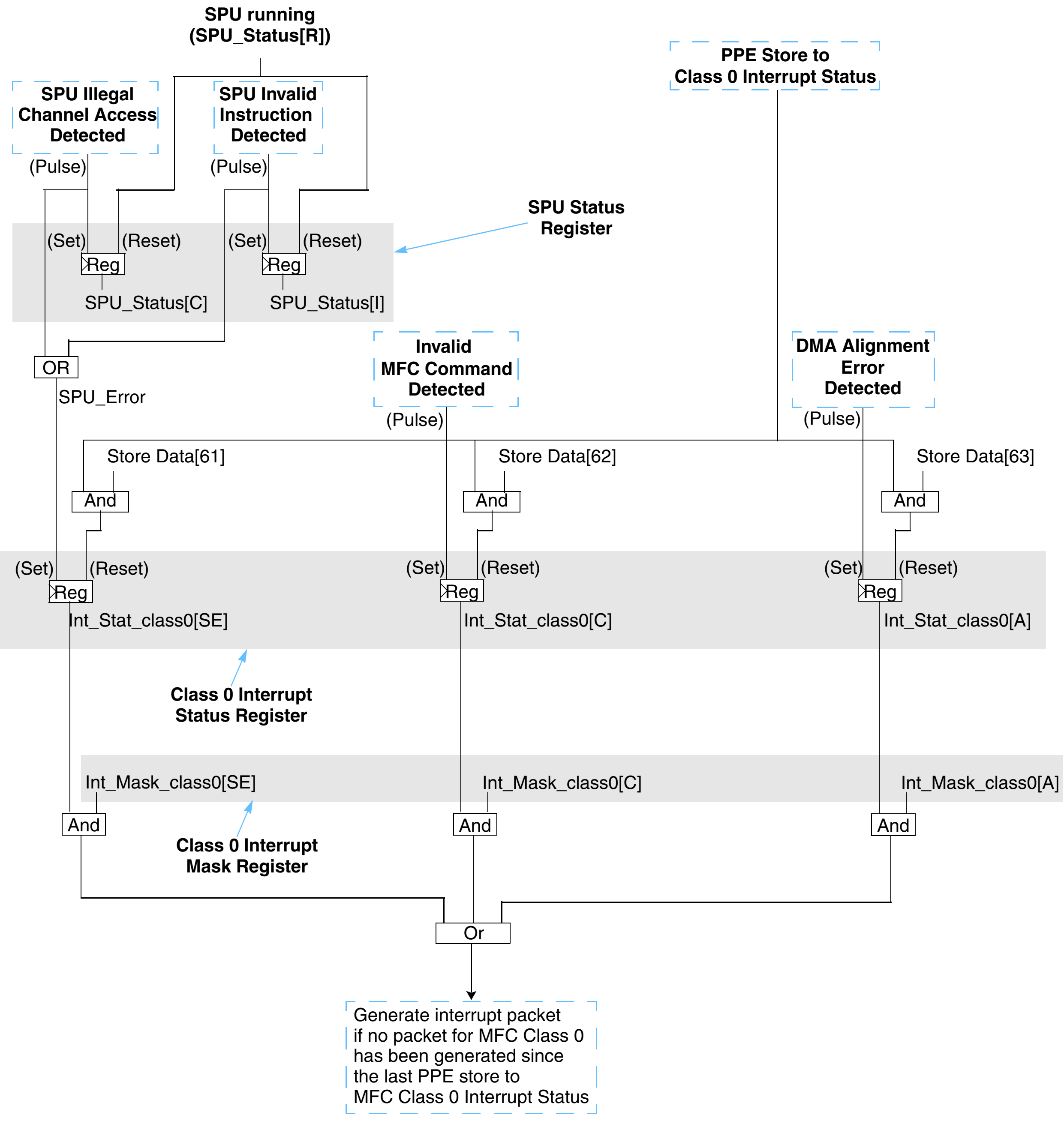
- If the SPU attempts an illegal channel access, the C bit of the SPU Status Register is set to ‘1’ and the SE bit is set to ‘1’.
- If the SPU attempts an invalid instruction, the I bit in SPU_Status is set to ‘1’ and the SE bit is set to ‘1’.

As shown in *Figure 21-2* on page 248, there are also two MFC exceptions that cause MFC class 0 interrupt status bits to be set:

- An invalid DMA command sets the C bit of the MFC Class 0 Interrupt Status Register to ‘1’.
- A DMA alignment error sets the A bit of the MFC Class 0 Interrupt Status Register to ‘1’.

Enabled bits in the MFC Class 0 Interrupt Status Register that are set to ‘1’ can cause a class 0 interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 260.

Figure 21-2. MFC Class 0 Interrupt Generation Process



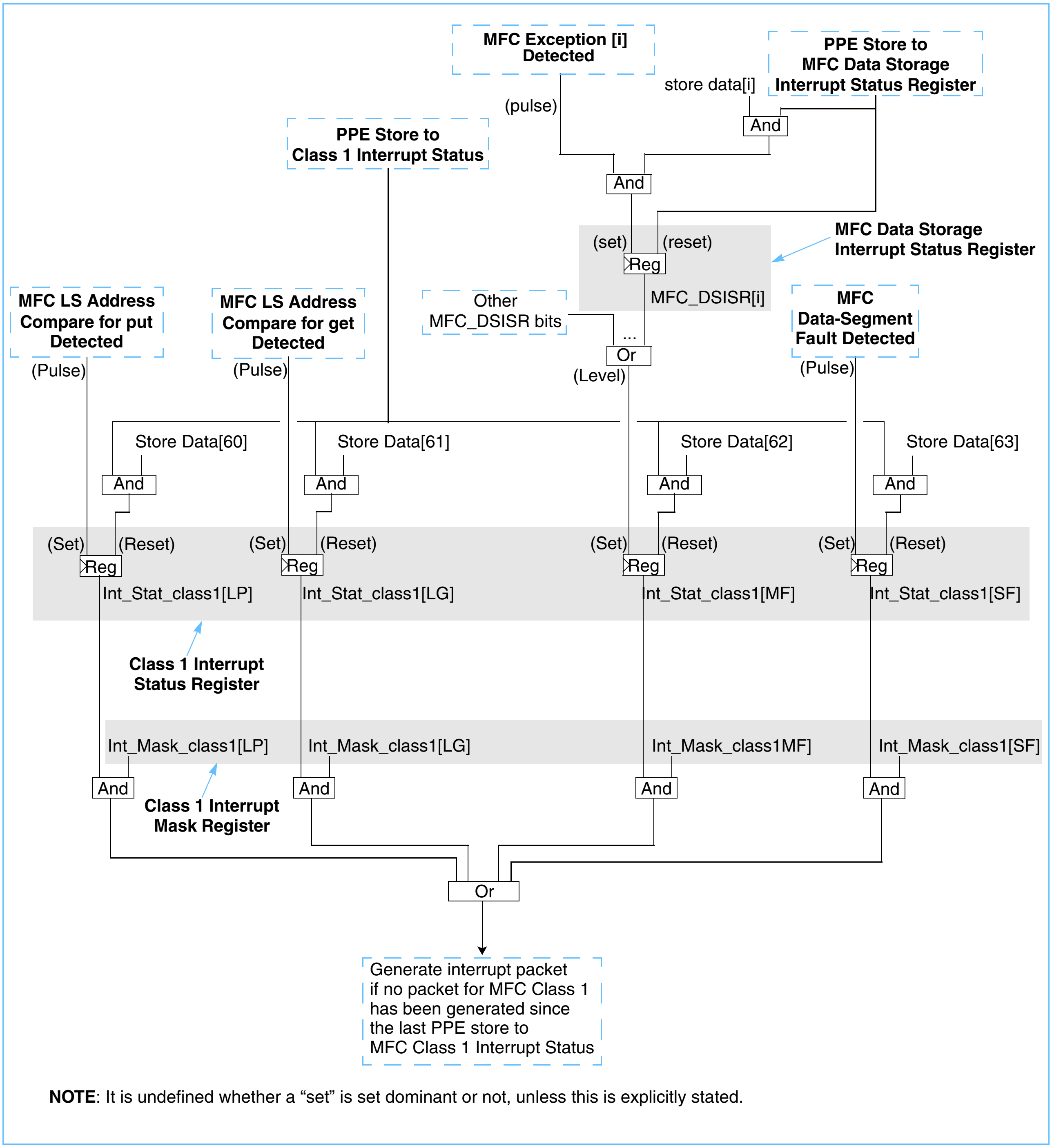
NOTE: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

21.5.2 Class 1 Interrupts

Only MFC events can cause class 1 interrupts to be generated, as shown in *Figure 21-3* on page 250.There are several defined MFC exceptions that set the bits of the MFC Data Storage Interrupt Status Register (see page 202) to ‘1’:

- If a bit in the MFC_DSISR is set to ‘1,’ the MF bit of the MFC Class 1 Interrupt Status Register (see page 258) is set to ‘1.’ In this case, the software interrupt handler needs to clear the bits in the MFC_DSISR by storing a ‘0’ in the respective bit position before attempting to reset bit ‘62’ of the MFC Class 1 Interrupt Status Register.
- If the MFC_DSISR bits are not cleared and the MF bit of the MFC Class 1 Interrupt Mask Register (see page 254) is set to ‘1’, another interrupt packet is generated.
- If an MFC data-segment fault sets the SF bit of the MFC Class 1 Interrupt Status Register to ‘1’.
- Enabled bits in the MFC Class 1 Interrupt Status Register that are set to ‘1’ can cause a class 1 interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 260.

Figure 21-3. MFC Class 1 Interrupt Generation Process


MFC Exception [i] Detected
PPE Store to MFC Data Storage Interrupt Status Register
store data[i]
(pulse)
And
PPE Store to Class 1 Interrupt Status
And
(set)
(reset)
Reg
MFC Data Storage Interrupt Status Register
MFC_DSISR[i]
Other MFC_DSISR bits
...
Or
(Level)
MFC LS Address Compare for put Detected
MFC LS Address Compare for get Detected
MFC Data-Segment Fault Detected
(Pulse)
(Pulse)
(Pulse)
Store Data[60]
Store Data[61]
Store Data[62]
Store Data[63]
And
And
And
And
(Set)
(Reset)
Reg
Int_Stat_class1[LP]
Int_Stat_class1[LG]
Int_Stat_class1[MF]
Int_Stat_class1[SF]
Class 1 Interrupt Status Register
Int_Mask_class1[LP]
Int_Mask_class1[LG]
Int_Mask_class1MF]
Int_Mask_class1[SF]
And
Class 1 Interrupt Mask Register
Or
Generate interrupt packet if no packet for MFC Class 1 has been generated since the last PPE store to MFC Class 1 Interrupt Status

NOTE: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

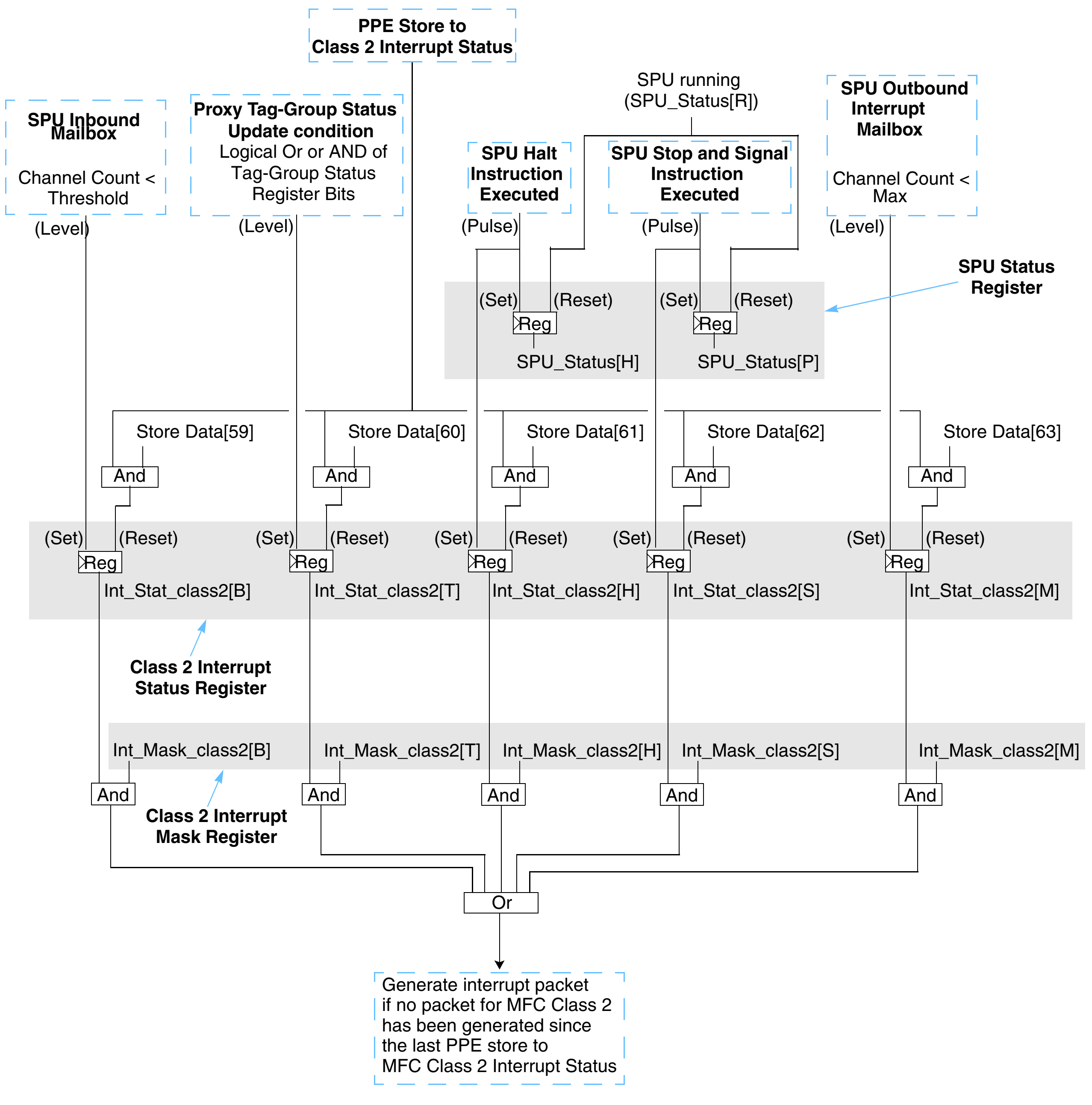
21.5.3 Class 2 Interrupts

Several SPU application actions cause class 2 interrupts to be generated. As shown in *Figure 21-4* on page 252:

- If amount of data in the SPU Inbound Mailbox queue falls below an implementation-dependent threshold, the B bit in the Class 2 Interrupt Status Register is set to '1'.
- If the Proxy Tag-Group Status Update condition is met, the T bit in the Class 2 Interrupt Status Register is set to '1'.
- If the SPU executes a halt instruction, the H bit in the SPU Status Register is set to ‘1’, and the H bit in the Class 2 Interrupt Status Register is set to ‘1’.
- If the SPU executes a stop-and-signal instruction, the P bit in SPU_Status is set to ‘1’, and the S bit in the MFC Class 2 Interrupt Status Register is set to ‘1’.
- If the SPU writes to the SPU Outbound Interrupt Mailbox Register (see page 213), the SPU Write Outbound Interrupt Mailbox Channel count is decremented. Whenever the SPU Write Outbound Interrupt Mailbox Channel count is less than the maximum supported by the implementation, the M bit in the MFC Class 2 Interrupt Status Register is set to ‘1’.
- Enabled bits in the MFC Class 2 Interrupt Status Register that are set to ‘1’ can cause a class 2 interrupt packet to be sent as described in *Section 21.8 Interrupt Routing Register* beginning on page 260.

The software interrupt handler can reset the class 2 MFC interrupt status bits without clearing SPU_Status. The SPU_Status bits are automatically reset when the SPU restarts.

Figure 21-4. MFC Class 2 Interrupt Generation Process


PPE Store to Class 2 Interrupt Status
SPU running (SPU_Status[R])
SPU Outbound Interrupt Mailbox
Channel Count < Max
SPU Inbound Mailbox
Channel Count < Threshold
Proxy Tag-Group Status Update condition
Logical Or or AND of Tag-Group Status Register Bits
SPU Halt Instruction Executed
SPU Stop and Signal Instruction Executed
(Level)
(Level)
(Pulse)
(Pulse)
(Level)
SPU Status Register
(Set)
(Reset)
Reg
SPU_Status[H]
SPU_Status[P]
Store Data[59]
Store Data[60]
Store Data[61]
Store Data[62]
Store Data[63]
And
Int_Stat_class2[B]
Int_Stat_class2[T]
Int_Stat_class2[H]
Int_Stat_class2[S]
Int_Stat_class2[M]
Class 2 Interrupt Status Register
Int_Mask_class2[B]
Int_Mask_class2[T]
Int_Mask_class2[H]
Int_Mask_class2[S]
Int_Mask_class2[M]
Class 2 Interrupt Mask Register
Or
Generate interrupt packet if no packet for MFC Class 2 has been generated since the last PPE store to MFC Class 2 Interrupt Status

Note: It is undefined whether a “set” is set dominant or not, unless this is explicitly stated.

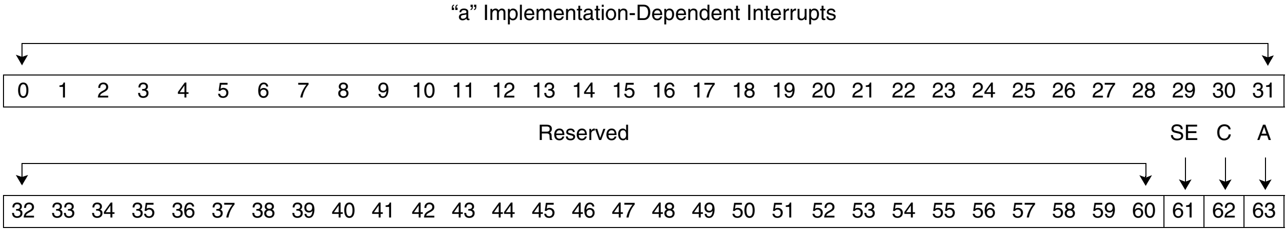
21.6 MFC Interrupt Mask Registers

There are three interrupt mask registers in each MFC: one for each class of interrupt (error, translation, application) as defined in *Section 21.4* on page 245. The interrupt mask registers allow privileged software to select which MFC and SPU events are allowed to generate an external interrupt to the PPE. Each bit in these registers has a corresponding status bit.

21.6.1 Class 0 Interrupt Mask Register (INT_Mask_class0)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0100’; where n is the SPE number

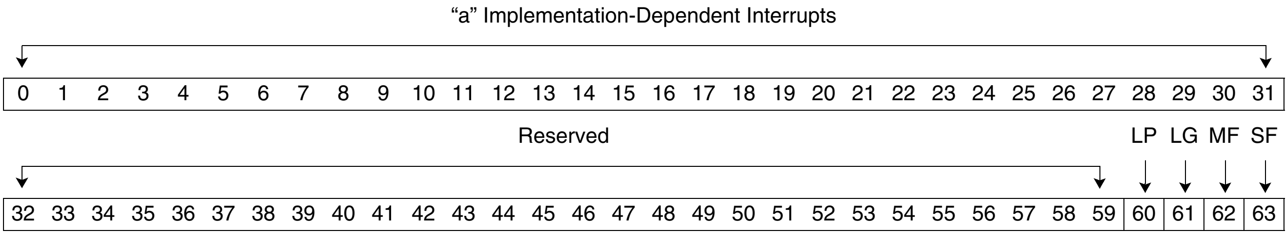


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled 1 Interrupt enabled
32:60	Reserved	Reserved.
61	SE	Enable for SPU error interrupt 0 Interrupt disabled 1 Interrupt enabled
62	C	Enable for invalid DMA command interrupt. 0 Interrupt disabled 1 Interrupt enabled
63	A	Enable for MFC DMA alignment interrupt. 0 Interrupt disabled 1 Interrupt enabled

21.6.2 Class 1 Interrupt Mask Register (INT_Mask_class1)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0108’; where n is the SPE number

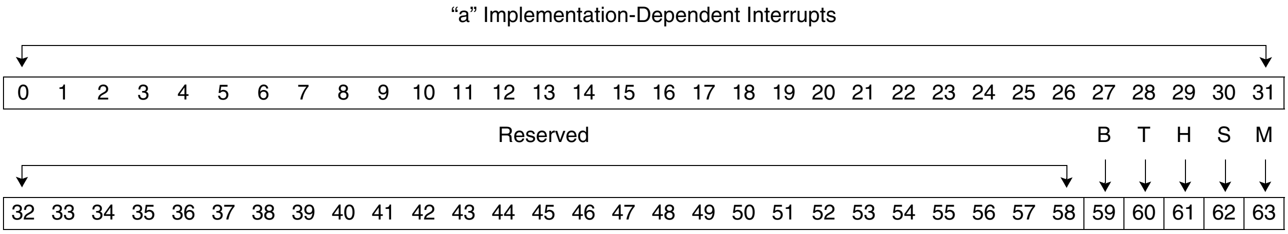


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled 1 Interrupt enabled
32:59	Reserved	Reserved.
60	LP	Enable for MFC local storage address compare suspend on **put** interrupt 0 Interrupt disabled 1 Interrupt enabled
61	LG	Enable for MFC local storage address compare suspend on **get** interrupt 0 Interrupt disabled 1 Interrupt enabled
62	MF	Enable for MFC data-storage interrupt (mapping fault). 0 Interrupt disabled 1 Interrupt enabled
63	SF	Enable for MFC data segment interrupt (segment fault). 0 Interrupt disabled 1 Interrupt enabled

21.6.3 Class 2 Interrupt Mask Register (INT_Mask_class2)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0110’; where n is the SPE number



Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Mask for implementation-dependent interrupts. 0 Interrupt disabled 1 Interrupt enabled
32:58	Reserved	Reserved.
59	B	Enable for SPU mailbox threshold interrupt. 0 Interrupt disabled 1 Interrupt enabled
60	T	Enable for DMA tag group completion. 0 Interrupt disabled 1 Interrupt enabled
61	H	Enable for SPU halt instruction trap or single instruction step complete. 0 Interrupt disabled 1 Interrupt enabled
62	S	Enable for SPU stop-and-signal instruction trap. 0 Interrupt disabled 1 Interrupt enabled
63	M	Enable for mailbox interrupt. 0 Interrupt disabled 1 Interrupt enabled

Note: Setting the threshold is implementation-dependent.

21.7 MFC Interrupt Status Registers

There are three interrupt status registers for each MFC: one for each class of interrupt (error, translation, application) as defined in *Section 21.4* on page 245. The interrupt status registers allow privileged software to determine which MFC and SPU events caused the external interrupt to be presented to the PPE. Refer to *PowerPC Architecture, Book III* for a description of how external interrupts are processed by the PPE.

When an interrupt event occurs, the corresponding interrupt status bit is set. Reads of an Interrupt Status Register are nondestructive. To reset an interrupt status bit, software must write a ‘1’ to the bit corresponding to the interrupt status. Writing a ‘0’ to any bit location does not change the state of the interrupt status. If software writes a ‘1’ to an interrupt status bit at the same time that hardware sets the bit due to an interrupt event, the resulting value in the Interrupt Status Register is undefined if the condition that set the interrupt status bit is a pulse. This is the case for MFC class 0 interrupt status bits 61 through 63, MFC class 1 interrupt status bit 63, and MFC class 2 interrupt status bits 61 and 62.

An interrupt packet is transmitted when the logical AND for the same class of the MFC Interrupt Status Register and the MFC Interrupt Mask Register is nonzero and an interrupt packet has not been transmitted since the last software write of the MFC Interrupt Status Register. Even though additional events can occur for a given class, only one interrupt packet for that class is sent until software stores to the MFC Interrupt Status Register for that class. After the store occurs, another interrupt packet is sent if the logical AND for the same class of the MFC Interrupt Status Register and the MFC Interrupt Mask Register is nonzero.

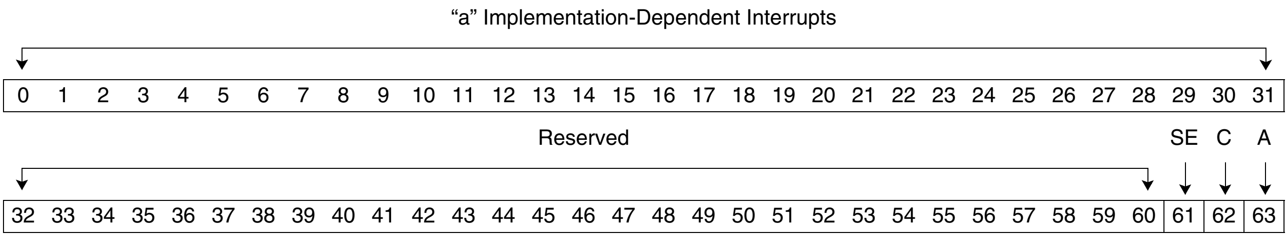
Implementation Note:

By the preceding definition, an interrupt packet could be transmitted if an interrupt event sets an MFC Interrupt Status Register bit to ‘1’ or if software resets an MFC Interrupt Mask Register bit to ‘0’.

Each bit in the MFC Interrupt Status Registers has a corresponding mask in the MFC Interrupt Mask Registers defined in *Section 21.6* on page 253.

21.7.1 Class 0 Interrupt Status Register (INT_Stat_class0)

Access Type	Read/Write
Base Address Offset	(BP_Base I P1(n)) + x‘0140’; where n is the SPE number

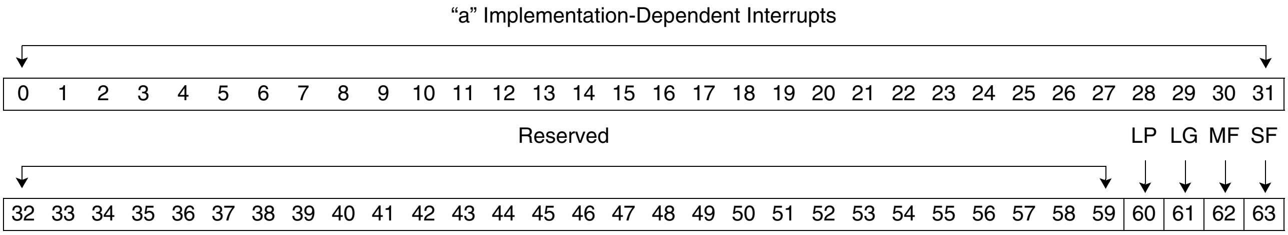


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
32:60	Reserved	Reserved
61	SE	Status for SPU error interrupt (see implementation-specific documentation for more details on errors) 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
62	C	Status for invalid DMA command interrupt 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
63	A	Status for DMA alignment interrupt 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type

21.7.2 Class 1 Interrupt Status Register (INT_Stat_class1)

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0148’; where n is the SPE number

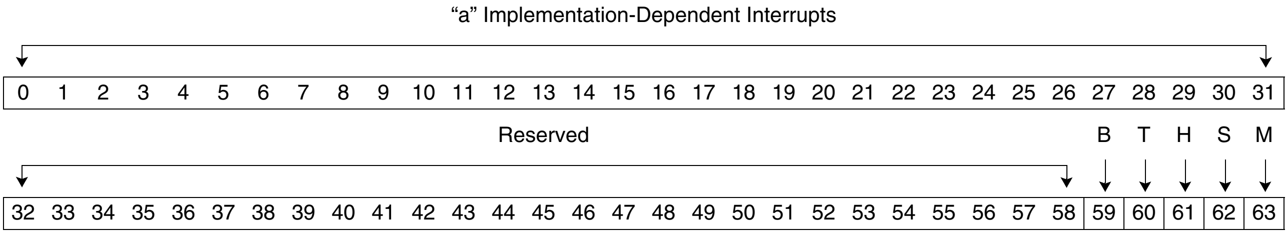


Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
32:59	Reserved	Reserved
60	LP	Status for MFC local storage address compare suspend on **put** interrupt 0 Interrupt disabled 1 Interrupt enabled
61	LG	Status for MFC local storage address compare suspend on **get** interrupt 0 Interrupt disabled 1 Interrupt enabled
62	MF	Status for MFC data storage interrupt (mapping fault) 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
63	SF	Status for MFC data segment interrupt (segment fault) 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type

21.7.3 Class 2 Interrupt Status Register (INT_Stat_class2)

Access Type Read/Write

Base Address Offset (BP_Base I P1(n)) + x‘0150’; where n is the SPE number



Bits	Field Name	Description
0:31	“a” Implementation-Dependent Interrupts	Status for implementation-dependent interrupts 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
32:58	Reserved	Reserved
59	B	Status for SPU Mailbox Threshold Interrupt 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
60	T	Status for SPU Tag-Group Complete Interrupt 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
61	H	Status for SPU Halt Instruction Trap or Single Instruction Step Complete 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
62	S	Status for SPU Stop-and-Signal Instruction Trap 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type
63	M	Status for Mailbox Interrupt 0 Interrupt not pending for corresponding interrupt type 1 Interrupt pending for corresponding interrupt type

21.8 Interrupt Routing Register (INT_Route)

The Interrupt Routing Register allows privileged software to select which PPE or which external interrupt controller services to interrupt. There is a set of routing information for each class of interrupt in the Interrupt Routing Register.

For each class, the register contains an 8-bit priority and an 8-bit interrupt destination. The interrupt destination indicates which logical PPE or external interrupt controller will be sent interrupt packets for MFC interrupts of the corresponding interrupt class.

The specific meaning of the priority and interrupt destination field is implementation-dependent. Refer to the specific implementation documentation for more information.

Access Type Read/Write

Base Address Offset (BP_Base I P1(n)) + x'0180'; where n is the SPE number

Class 0 Priority								Class 0 Interrupt Destination								Class 1 Priority								Class 1 Interrupt Destination							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

Class 2 Priority								Class 2 Interrupt Destination								Reserved															
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

Bits	Field Name	Description
0:7	Class 0 Priority	Interrupt priority for class 0 interrupts. The value written to this field will be used as the priority for all class 0 interrupts from the associated SPE.
8:15	Class 0 Interrupt Destination	Indicates which logical PPE or external interrupt controller will be sent interrupt packets for MFC class 0 interrupts.
16:23	Class 1 Priority	Interrupt priority for class 1 interrupts. The value written to this field will be used as the priority for all class 1 interrupts from the associated SPE.
24:31	Class 1 Interrupt Destination	Indicates which logical PPE or external interrupt controller will be sent interrupt packets for MFC class 1 interrupts.
32:39	Class 2 Priority	Interrupt priority for class 2 interrupts. The value written to this field will be used as the priority for all class 2 interrupts from the associated SPE.
40:47	Class 2 Interrupt Destination	Indicates which logical PPE or external interrupt controller will be sent interrupt packets for MFC class 2 interrupts.
48:63	Reserved	Reserved

22. Power Management

The Cell Broadband Engine Architecture (CBEA) defines five major states for power management: Active, Slow (n), Pause (n), State Retained and Isolated (SRI), and State Lost and Isolated (SLI). Typically, the more aggressive the power management state, the more time is required to enter and exit the state (see *Table 22-1* to determine which states are more aggressive). These states apply to the various components of a CBEA-compliant processor as well as the full processor. The intent is to allow each component within a CBEA-compliant processor to be set to the same or different power-management states.

Multiple power-management states provide privileged software with the control necessary to manage the power of a CBEA-compliant processor while meeting the real-time demands of the system. Implementation of the various states, and the associated power savings, are implementation-dependent. Refer to the specific implementation documentation for more information.

Table 22-1. Power Management States

Power Management State	Power Savings	Description	Notes
Active	Less aggressive	In active state, the performance of a component is not limited by power management.	
Slow (n)	↓	In the slow (n) state, performance can be degraded for power savings. A higher value of n indicates a more aggressive power savings, and potential for more performance degradation. Real-time performance can be limited. The functionality of the component is not altered.	1
Pause (n)		In the pause (n) state, the component is not guaranteed to make forward progress. The component state is maintained as well as the system integrity. A higher value of n indicates a more aggressive power savings, and potential for more performance degradation.	1
State Retained and Isolated (SRI)		In the SRI state, all access to the component is inhibited. The state remaining on the component is retained. The component must be prepared by privileged software or hardware to maintain system integrity. The component does not make forward progress.	
State Lost and Isolated (SLI)	More aggressive	In the SLI state, the component is effectively removed from the system. The state of the component is not retained, and the component will not respond to a system event.	

1. Where *n* in the slow and pause states denotes a higher degree of power savings.

Typically, the transition between the various states is controlled by privileged software. In some cases, an implementation can allow events to alter the power-management state of a CBEA-compliant processor.

23. Version Control

The CBEA consists of several components, which can have different version levels or revisions. To allow for this, the CBEA provides a version register for each component and an overall version register referred to as the CBEA-Compliant Processor Version Register.

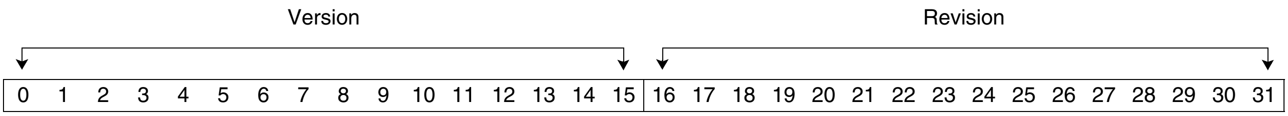
23.1 CBEA-Compliant Processor Version Register (BP_VR)

The CBEA-Compliant Processor Version Register is a 32-bit read only register that contains values that identify the version and revision level of the CBEA-compliant processor. The contents of the CBEA-Compliant Processor Version Register are only accessible using a PPE move-from special-purpose register (**mfspr**) instruction. Read access to the CBEA-Compliant Processor Version Register is privileged; while write access is not provided. There is only one CBEA-Compliant Processor Version Register per CBEA-compliant processor.

Version numbers are assigned by the CBEA process. Revision numbers are assigned by an implementation-defined process.

Access Type Read Only

PPE SPR Number x‘3FE’



Bits	Field Name	Description
0:15	Version	A 16-bit number that identifies the version of the processor. Different version numbers indicate major differences between processors, such as which optional facilities and instructions are supported.
16:31	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between processors with the same version number, such as clock rate and engineering change level.

Implementation Note:

Although the classification of differences between CBEA-compliant processors as “major” or “minor” is somewhat arbitrary, the following are examples of differences that generally should be considered “major:”

- Number and type of execution units
- Optional facilities, instructions, and commands supported
- Level of instruction and command support (hard-wired or emulated)
- Size, geometry, and management of caches, TLBs, and SPU local storage

The following are examples of differences that generally should be considered “minor:”

- Remapping a processor to a new technology
- Redesigning a critical path to increase clock rate
- Fixing bugs

In general, any change to a CBEA-compliant processor should cause a new BP_VR value to be assigned. Any change to a CBEA-compliant processor, even a change that is not expected to be apparent to software, should cause a new revision number to be assigned in case the change has introduced an error that software must circumvent.

23.2 PPE Processor Version Register (PVR)

PowerPC Architecture, Book III describes a processor version register, which contains a 32-bit value that identifies the specific version (model) and revision level of the PPE portion of the CBEA.

There is one PVR for each PPE in a CBEA-compliant processor. The contents of the PPE Processor Version Register are only accessible using a PPE move-from special-purpose register (**mfspr**) instruction.

Version numbers are assigned by the PowerPC Architecture process. Revision numbers are assigned by an implementation-defined process.

Refer to *PowerPC Architecture, Book III* for a complete description of the PVR.

23.3 SPU Version Register (SPU_VR)

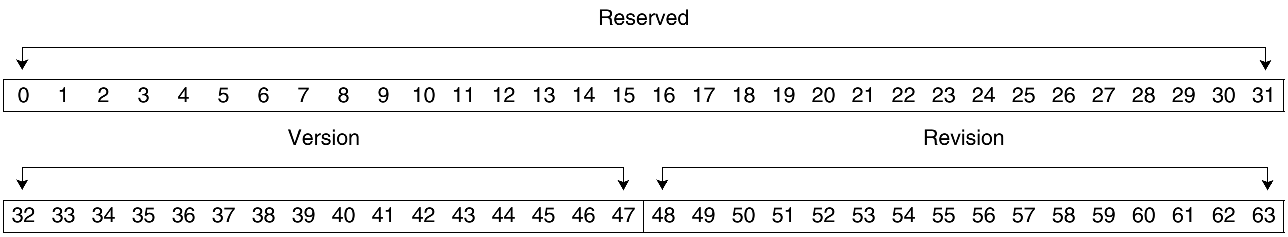
The SPU Version Register contains a 32-bit value that identifies the specific version (model) and revision level of the SPU portion of the CBEA. The contents of this register are accessible from the PPE using a load doubleword (**ld**) instruction. Read access to the SPU Version Register from the PPE is privileged. Write access is not provided. Access to this register from the SPU is not provided. There is one SPU Version Register for each SPU in a CBEA-compliant processor.

Version numbers are assigned by the SPU architecture process. Revision numbers are assigned by an implementation-defined process.

The SPU_VR distinguishes between processors that differ in attributes that can affect software. It contains two fields.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0020’, where n is the SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:47	Version	A 16-bit number that identifies the version of the SPU. Different version numbers indicate major differences between SPUs, such as which optional facilities and instructions are supported.
48:63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between SPUs with the same version number, such as clock rate and engineering change level.

23.4 MFC Version Register (MFC_VR)

The MFC Version Register contains a 32-bit value that identifies the version and revision level of the MFC component of the CBEA. The contents of the MFC Version Register are accessible using a PPE **ld** instruction.

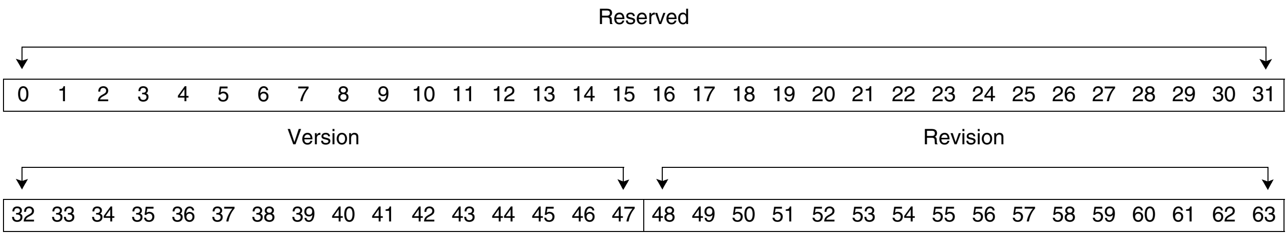
Read access to the MFC_VR is privileged; write access is not provided. There is one MFC Version Register for each MFC in the CBEA-compliant processor.

Version numbers are assigned by the CBEA process. Revision numbers are assigned by an implementation-defined process, described in the specific implementation documentation.

The MFC_VR distinguishes between processors that differ in attributes that can affect software. It contains two fields.

Access Type Read Only

Base Address Offset (BP_Base | P1(n)) + x‘0018’, where n is the SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:47	Version	A 16-bit number that identifies the version of the MFC. Different version numbers indicate major differences between MFC units, such as which optional facilities and instructions are supported.
48:63	Revision	A 16-bit number that distinguishes between implementations of the version. Different revision numbers indicate minor differences between MFC units with the same version number, such as clock rate and engineering change level.

23.5 SPU Identification Register (SPU_IDR)

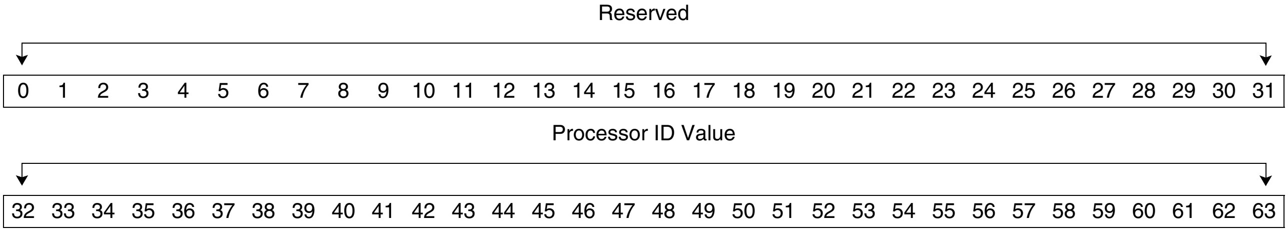
The SPU Identification Register contains a 32-bit value that can be used to distinguish the SPU from other SPUs in the system. The contents of this register are accessible from the PPE using an **ld** instruction. Read access to the SPU Identification Register from the PPE is privileged. Write access, if provided, is implementation-dependent. (Refer to the specific implementation documentation for more information.)

Access to this register from the SPU is not provided. There is one SPU Identification Register for each SPU in the CBEA-compliant processor.

SPU Identification Register initialization is implementation-dependent. Refer to the specific implementation documentation for more information.

Access Type Read/Write

Base Address Offset (BP_Base | P1(n)) + x‘0010’, where n is the SPE number.



Bits	Field Name	Description
0:31	Reserved	Set to zeros.
32:63	Processor ID Value	Used to distinguish the SPU from other SPUs in the system.

Appendix A. Memory Maps

This appendix contains the mapping of all registers defined by the Cell Broadband Engine Architecture in the real address space. The memory map for a single CBEA-compliant processor is divided into six sections: the SPU local storage area, the SPE privileged area, the PPE area, the Internal Interrupt Controller (IIC) area, the Power Management and Debug Area (PMD) area, and an Implementation-Dependent Expansion Area (IDEA). The size of each area is based on the number of units (PPEs, SPUs, and IICs) in the CBEA-compliant processor.

The starting location for the memory map is defined in an implementation-dependent base address register (BP_Base). The BP_Base is used to calculate the address for all registers. An implementation can place address holes in the memory map (that is, areas where registers are not defined) to simplify decoding of the registers.

The base address register (BP_Base) for a CBEA-compliant processor is implementation-dependent. Refer to the specific implementation documentation to determine how the base registers are initialized.

Table A-1. Broadband Processor Memory Map (Page 1 of 2)

Real Address (Hexadecimal)	Area	Description
SPE Local Storage (LS(n)), Problem-State (PS(n)), and Privilege 2 Area (P2(n)); where 0 ≤ n ≤ (number of SPEs -1)[2] [LS(0) = BP_Base[1]; Starting address of area] [spe_area_size = (2 * LS_size) rounded up to a power of 2 boundary]		
Local Storage Area of SPE(n); where LS(n) = n * spe_area_size		
BP_Base | LS(n)	LS(n) area	Start of local storage area for SPU(n)
BP_Base | LS(n) + LS_Size - 1	End of LS(n) area	End of local storage area
Problem-State of SPE(n); where PS(n) = LS(n) + (spe_area_size >> 1) [6]		
BP_Base | PS(n)	SPE(n) problem state area	Start of problem-state area for SPE group(n)
BP_Base | PS(n) + x'1FFFF	End of PS(n) area	End of local storage area
Privilege 2 Area of SPE(n); where P2(n) = PS(n) + x'20000		
BP_Base | P2(n)	SPE(n) Privilege 2 area	Start of Privilege 2 area for SPE(n)
BP_Base | P2(n) + x'1FFFF'	End of SPE(n) area	End of SPE(n) area
SPE Privileged Area (P1(n)) 1 (P1); where 0 ≤ n < number of SPEs [2] [P1(0) ≥ P2(max) + 0x20000; Starting address of area] [P1(n) = P1(0) + (n * 0x2000); where 0 ≤ n ≤ (number of SPEs - 1)]		
BP_Base | P1(n)	SPE(n) Privilege 1 area	Start of Privilege 1 area for SPE(n)
BP_Base | P1(n) + x'1FFF'	End of SPE(n) area	End of SPE(n) Privilege 1 area

Notes:

1. This table assumes that the base starts on a power of 2 boundary which is greater than or equal to the size of the memory map area.
2. The value "n" ranges from zero to the number of SPEs minus 1 (0 ≤ n ≤ (number of SPEs - 1)). If the number of SPEs is not a power of 2, an implementation can choose to increase the value of "n" to the next power of 2 boundary and reserve the extra space. Doing so simplifies the decode of the address ranges.
3. The value "p" ranges from zero to the number of PPEs minus 1 (not threads) (0 ≤ p < [number of PPEs - 1]). If the number of PPEs is not a power of 2, an implementation can choose to increase the value of "p" to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
4. The value "t" ranges from zero to the number of threads in the associated PPE minus 1 (0 ≤ t < [number of PPEs threads - 1]).
5. Memory map pad (MM_pad) is used to pad the Broadband Processor memory-map area to at least a 64-KB boundary.
6. The symbol >> indicates shift by one to the right.

Table A-1. Broadband Processor Memory Map (Page 2 of 2)

<table>
<tr><th>Real Address
(Hexadecimal)</th><th>Area</th><th>Description</th></tr>
<tr><td colspan="3">**PowerPC Area(p);** where 0 ≤ p < number of real PowerPC > 1 (**PPEs**; not **PPE** threads)[3]
[PPE(0) ≥ P1(max) + 0x2000; Starting address of area]
[PPE(p) = PPE(0) + (p * 0x1000); where 0 ≤ n ≤ (number of SPEs - 1)]</td></tr>
<tr><td>BP_Base | PPE(p)</td><td>PPE(p) privileged area</td><td>Start of privileged area for PPE(p)</td></tr>
<tr><td>BP_Base | PPE(p) + x‘FFF’</td><td>End of PPE(p) area</td><td>End of PPE(p) area</td></tr>
<tr><td colspan="3">**Internal Interrupt Controller Area(p);** where 0 ≤ p < number of physical PowerPC Cores -1 (**PPEs**; not **PPE** threads)[3]
[IIC(0) ≥ PPE(max) + 0x1000; Starting address of area]
[IIC(p) = IIC(0) + (p * x’1000’)]
(Note: If more than 16 threads per PPE are needed, additional IIC areas are added and the threads are divided between the areas</td></tr>
<tr><td>BP_Base | IIC(p)</td><td>IIC implementation dependent area</td><td>Start of internal-interrupt controller implementation dependent area</td></tr>
<tr><td>BP_Base | IIC(p) + x‘3FF’</td><td>IIC implementation dependent area</td><td>End of internal-interrupt controller implementation dependent area</td></tr>
<tr><td colspan="3">**Internal Interrupt Controller Thread Control Block(t);** where 0 ≤ t < number of PowerPC threads -1[4]
[IIC_TCB(t) = IIC(p) + x’400’ + (t * x’20’)]</td></tr>
<tr><td>BP_Base | IIC_TCB(t)</td><td>Interrupt control blocks</td><td>Start of internal-interrupt controller’s area (see *Section 21.3 Internal Interrupt Controller Registers* on page 239)</td></tr>
<tr><td>BP_Base | IIC_TCB(t) + x‘3FFF’</td><td>End of IIC_TCB area</td><td>End of internal-interrupt controller thread control block area</td></tr>
<tr><td>BP_Base | IIC(p) + x‘FFF’</td><td>End of IIC area</td><td>End of internal-interrupt controller area</td></tr>
<tr><td colspan="3">**Power Management and Debug Area (PMD)**
[PMD ≥ IIC(max) + 0x1000; Starting address of area]</td></tr>
<tr><td>BP_Base | PMD</td><td>Power management debug area</td><td>MMIO area for power management and architected performance monitor and debug features</td></tr>
<tr><td>BP_Base | PMD + x‘FFF’</td><td>End of power management debug area</td><td>End of power management and architecture performance monitor and debug area</td></tr>
<tr><td colspan="3">**Implementation-Dependent Expansion Area (IDEA)**
[IDEA ≥ PMD + 0x1000; Starting address of area]</td></tr>
<tr><td>BP_Base | IDEA</td><td>Beginning of Implementation-dependent area</td><td>MMIO area for implementation-dependent features are defined in the specific implementation documentation.</td></tr>
<tr><td>BP_Base | IDEA + MM_pad[5]</td><td>End of implementation-dependent area</td><td>End of MMIO area for implementation-dependent features</td></tr>
</table>

Notes:

1. This table assumes that the base starts on a power of 2 boundary which is greater than or equal to the size of the memory map area.
2. The value “n” ranges from zero to the number of SPEs minus 1 (0 ≤ n ≤ (number of SPEs - 1)). If the number of SPEs is not a power of 2, an implementation can choose to increase the value of “n” to the next power of 2 boundary and reserve the extra space. Doing so simplifies the decode of the address ranges.
3. The value “p” ranges from zero to the number of PPEs minus 1 (not threads) (0 ≤ p < [number of PPEs - 1]). If the number of PPEs is not a power of 2, an implementation can choose to increase the value of “p” to the next power of 2 boundary and reserve the extra space. Doing so simplifies decoding the address ranges.
4. The value “t” ranges from zero to the number of threads in the associated PPE minus 1 (0 ≤ t < [number of PPEs threads - 1]).
5. Memory map pad (MM_pad) is used to pad the Broadband Processor memory-map area to at least a 64-KB boundary.
6. The symbol >> indicates shift by one to the right.

Implementation Note:

A Broadband Processor implementation must provide at least one base register for relocating the internal registers. All base register (BP_Base) values must be initialized during the CBEA-compliant processor initialization sequence. The base register can either be set using a program sequence or initialized using a hardware method such as scan or JTAG.

A.1 SPE Problem State Memory Map

Table A-2 shows how the SPU problem state registers are mapped into the real address space of the system. Some registers are defined as byte and half-word quantities, but all accesses to these registers are required to be a minimum of 32 bits. (The descriptions are also working hyperlinks, so if you are viewing the document on-line, you can click on them to “jump” to the relevant page.)

Table A-2. SPE Problem State Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
Multiisource Synchronization Area			
x‘00000’	MFC_MSSync	MFC Multisource Synchronization Register (see page 96)	Read/Write
Command Parameter Area			
x‘03000’	Reserved	Reserved for future expansion	Reserved
x‘03004’	MFC_LSA	MFC Local Storage Address Register (see page 75)[1]	Write Only
x‘03008’	MFC_EAH	MFC Effective Address High Register (see page 76)[1]	Write Only
x‘0300C’	MFC_EAL	MFC Effective Address Low Register (see page 77)[1]	Write Only
x‘03010’	MFC_Size	MFC Transfer Size Register (see page 74)[1, 3] (Upper 16 bits of register)	Write Only
	MFC_Tag	MFC Command Tag Register (see page 73)[1, 3] (Lower 16 bits of register)	Write Only
x‘03014’	MFC_ClassID	MFC Class ID Register (see page 72)[1, 2] (Upper 16 bits of register for write)	Write Only
	MFC_CMD	MFC Command Opcode Register (see page 71) [1, 3] (Lower 16 bits of register for write)	Write Only
	MFC_CMDStatus	MFC Command Status Register (see page 80)(all 32-bits for read)	Read Only
MFC Command Queue Control Area			
x‘03020’:x‘030FF’	Reserved	Reserved	
x‘03104’	MFC_QStatus	MFC Queue Status Register (see page 81)	Read Only
x‘03204’	Prxy_QueryType	Proxy Tag-Group Query Type Register (see page 83)	Read/Write
x‘0321C’	Prxy_QueryMask	Proxy Tag-Group Query Mask Register (see page 84)	Read/Write
x‘0322C’	Prxy_TagStatus	Proxy Tag-Group Status Register (see page 85)	Read Only
x‘03330’:x‘03FFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_ClassID and MFC_CMD registers must be written with a single 32-bit store instruction.
3. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.

Table A-2. SPE Problem State Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
SPU Control Area			
x‘04004’	SPU_Out_Mbox	SPU Outbound Mailbox Register (see page 91)	Read Only
x‘0400C’	SPU_In_Mbox	SPU Inbound Mailbox Register (see page 92)[1]	Write Only
x‘04014’	SPU_Mbox_Stat	SPU Mailbox Status Register (see page 93)	Read Only
x‘0401C’	SPU_RunCntl	SPU Run Control Register (see page 86)	Read/Write
x‘04024’	SPU_Status	SPU Status Register (see page 87)	Read Only
x‘04034’	SPU_NPC	SPU Next Program Counter Register (see page 89)	Read/Write
x‘04038’:x‘13FFF’	Reserved	Reserved	
Signal-Notification Area			
x‘1400C’	SPU_Sig_Notify_1	SPU Signal Notification 1 Register (see page 94)	Read/Write
x‘14010’:x‘1BFFF’	Reserved	Reserved	
x‘1C00C’	SPU_Sig_Notify_2	SPU Mailbox Status Register (see page 93)	Read/Write
x‘1C010’:x‘1FFFF’	Reserved	Reserved	

1. Reading of these registers should be allowed for diagnostic purposes.
2. Both the MFC_ClassID and MFC_CMD registers must be written with a single 32-bit store instruction.
3. Both the MFC_Size and MFC_TAG registers must be written with a single 32-bit store instruction.

A.2 SPE Privilege 1 Memory Map

Table A-3 contains a list of all CBEA-compliant SPE registers, which allow only Privilege 1 access. For information on each of the registers, see the relevant page. (The descriptions are also working hyperlinks, so if you are viewing the document on-line, you can click on them to “jump” to the relevant page.)

Table A-3. SPE Privilege 1 Memory Map (Page 1 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Control and Configuration Area			
x‘0000’	MFC_SR1	MFC State Register One (see page 197)	Read/Write
x‘0008’	MFC_LPID	MFC Logical Partition ID Register (see page 199)	Read/Write
x‘0010’	SPU_IDR	SPU Identification Register (see page 267)	Read/Write
x‘0018’	MFC_VR	MFC Version Register (see page 266)	Read Only
x‘0020’	SPU_VR	SPU Version Register (see page 265)	Read Only
x‘0128’:x‘00FF’	Reserved	Reserved	
Interrupt Area			
x‘0100’	INT_Mask_class0	Class 0 Interrupt Mask Register (see page 253)	Read/Write
x‘0108’	INT_Mask_class1	Class 1 Interrupt Mask Register (see page 254)	Read/Write
x‘0110’	INT_Mask_class2	Class 2 Interrupt Mask Register (see page 255)	Read/Write
x‘0118’:x‘013F’	Reserved	Reserved	Reserved
x‘0140’	INT_Stat_class0	Class 0 Interrupt Status Register (see page 257)	Read/Write
x‘0148’	INT_Stat_class1	Class 1 Interrupt Status Register (see page 258)	Read/Write
x‘0150’	INT_Stat_class2	Class 2 Interrupt Status Register (see page 259)	Read/Write
x‘0158’:x‘017F’	Reserved	Reserved	Reserved
x‘0180’	INT_Route	Interrupt Routing Register (see page 260)	Read/Write
x‘0188’:x‘01FF’	Reserved	Reserved	Reserved
Atomic Unit Control Area			
x‘0200’	MFC_Atomic_Flush	MFC Atomic Flush Register (see page 212) This is an implementation-dependent register	Read/Write
x‘0208’:x‘03FF’	SPU_Cache_ImplRegs	SPU cache hardware implementation-dependent registers. Refer to the specific implementation documentation.	

1. An implementation should support reading of these registers for diagnostic purposes.

Table A-3. SPE Privilege 1 Memory Map (Page 2 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
Translation Lookaside Buffer (TLB) Management Registers			
x‘0400’	MFC_SDR	MFC Storage Description Register (see page 200) Also see the *PowerPC Architecture, Book III* for a description of this register.	Read/Write
x‘0408’:x‘04FF’	Reserved	Reserved	
x‘0500’	MFC_TLB_Index_Hint	TLB Index Hint Register (see page 186) Index of best TLB entry to update.	Read Only
x‘0508’	MFC_TLB_Index	TLB Index Register (see page 187) Index to TLB entry to update with TLB Real Page Number Register and TLB Virtual Page Number Register[1]	Write Only
x‘0510’	MFC_TLB_VPN	TLB Virtual Page Number Register (see page 188) Access to upper portion of TLB entry	Read/Write
x‘0518’	MFC_TLB_RPN	TLB Real Page Number Register (see page 189) Access to lower portion of TLB entry	Read/Write
x‘0520’:x‘053F’	Reserved	Reserved	
x‘0540’	MFC_TLB_Invalidate_Entry	TLB Invalidate Entry Register (see page 190) Virtual Page Number of TLB entry to invalidate[1] **Note:** Not available for PowerPC Processor Element (PPE).	Write Only
x‘0548’	MFC_TLB_Invalidate_All	TLB Invalidate All Register (see page 192) Invalidate all TLB entries (optional)[1] **Note:** Not available for PowerPC Processor Element (PPE).	Write Only
x'0550':'057F	Reserved	Reserved	’
(Memory Management. Implementation-dependent area: Refer to the specific implementation documentation.)			
x‘0580’:x‘05FF’	SPE_MMU_ImplRegs	SPE Memory Management Unit (MMU) Registers Refer to the specific implementation documentation for a description of this register.	
MFC Status and Control Area			
x‘0600’	MFC_ACCR	MFC Address Compare Control Register (see page 203)	Read/Write
x‘0610’	MFC_DSISR	MFC Data Storage Interrupt Status Register (see page 202)	Read/Write
x‘0620’	MFC_DAR	MFC Data Address Register (see page 201)	Read/Write
x‘0628’:x‘06FF’	Reserved	Reserved	
Replacement Management Table Area (RMT) (Implementation-dependent area. Refer to the specific implementation documentation)			
x‘0700’	MFC_TLB_RMT_Index	RMT Index Register (see page 233) Index of the replacement management tables	Read/Write
x‘0710’	MFC_TLB_RMT_Data	RMT Data Register (see page 234) Doubleword of RMT data pointed to by the RMT Index Register Entry contents are implementation-dependent.	Read/Write
x‘0718’:x‘07FF’	SPE_RMT_ImplRegs	SPE RMT hardware implementation-dependent registers (Same address listed with a different offset under memory management.)	

[1] An implementation should support reading of these registers for diagnostic purposes.

Table A-3. SPE Privilege 1 Memory Map (Page 3 of 3)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Command Data Storage Interrupt Area			
x‘0800’	MFC_DSIPR	MFC Data Storage Interrupt Pointer Register (see page 208) Contains a pointer to the command in the MFC command queue that caused the error condition.	Read Only
x‘0808’	MFC_LSACR	MFC Local Storage Address Compare Register (see page 205) 64-bit MFC Local Storage Address Compare Register	Read/Write
x‘0810’	MFC_LSCRR	MFC Local Storage Compare Result Register (see page 206) 64-bit MFC Local Storage Compare Results Register	Read Only
x‘0818’:x‘08FF’	Reserved	Reserved	
Real-Mode Support Registers			
x‘0900’	MFC_RMAB	MFC Real-Mode Address Boundary Register (see page 194)	Read/Write
x‘0908’:x‘0BFF’	Reserved	Reserved	
MFC Command Error Area			
x‘0C00’	MFC_CER	MFC Command Error Register (see page 207) Contains a pointer to the command in the DMA queue that caused the error condition.	Read Only
x‘0C08’:x‘0FFF’	Reserved	Reserved	
(Implementation-Dependent Area Refer to the specific implementation documentation for a detailed description of these registers)			
x‘1000’:x‘1FFF’	PV1_ImplRegs	Privilege 1 implementation-dependent registers	

1. An implementation should support reading of these registers for diagnostic purposes.

A.3 SPE Privilege 2 Memory Map

Table A-4 contains a list of all CBEA-compliant registers, which allow only Privilege 2 access. For information on each of the registers, see the relevant page. (The descriptions are also working hyperlinks, so if you are viewing the document on-line, you can click on them to “jump” to the relevant page.)

Table A-4. SPE Privilege 2 Memory Map (Page 1 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
MFC Registers			
x‘00000’:x‘010FF’	Reserved	Reserved	Reserved
Segment Lookaside Buffer Management Registers			
x‘01100’	Reserved	Reserved	Reserved
x‘01108’	SLB_Index	SLB Index Register (see page 179) Index of SLB entry to be updated by SLB_VSID and SLB_ESID ports[1]	Write Only
x‘01110’	SLB_ESID	SLB Effective Segment ID Register (see page 180) Access to the upper portion of an SLB entry	Read/Write
x‘01118’	SLB_VSID	SLB Virtual Segment ID Register (see page 181) Access to the lower portion of an SLB entry	Read/Write
x‘01120’	SLB_Invalidate_Entry	SLB Invalidate Entry Register (see page 182) ESID of SLB entry to invalidate[1]	Write Only
x‘01128’	SLB_Invalidate_All	SLB Invalidate All Register (see page 183) Invalidate all SLB entries[1]	Write Only
x‘01130’:x‘01FFF’	Reserved	Reserved	Reserved
Context-Save and Restore Area (Implementation Dependent Area: Refer to the specific implementation documentation.)			
x‘02000’:x‘02FFF’	MFC_CSR_ImplRegs	MFC Context-Save and Restore registers	
MFC Control			
x‘03000’	MFC_CNTL	MFC Control Register (see page 209)	Read/Write
x‘03008’:x‘03FFFF’	MFC_Cntl1_ImplRegs	Implementation-dependent control registers. Refer to the specific implementation documentation.	
Interrupt Mailbox			
x‘04000’	SPU_OutIntrMbox	SPU Outbound Interrupt Mailbox Register (see page 213) SPU writes, PPE reads.	Read Only
SPU Control			
x‘04040’	SPU_PrivCntl	SPU Privileged Control Register (see page 215)	Read/Write
x‘04058’	SPU_LSLR	SPU Local Storage Limit Register (see page 217)	Read/Write
x‘04060’	SPU_ChnlIndex	SPU Channel Index Register (see page 218) This register selects which SPU channel in the specified SPU(n) is accessed using the SPU Channel Count Register or SPU Channel Data Register.	Read/Write
x‘04068’	SPU_ChnlCnt	SPU Channel Count Register (see page 220) This register is used to read or to initialize the SPU Channel Count Register selected by the SPU Channel Index Register.	Read/Write

[1] An implementation should support reading of these registers for diagnostic purposes

Table A-4. SPE Privilege 2 Memory Map (Page 2 of 2)

Offset (Hexadecimal)	Register	Description	Access Type
x'04070'	SPU_ChnlData	SPU Channel Data Register (see page 219) This register is used to read or to initialize the SPU channel data selected by the SPU Channel Index Register.	Read/Write
x'04078'	SPU_Cfg	SPU Configuration Register (see page 221) This register is used to read or set the configuration of the SPU Signal-Notification Registers in the specified SPU(n).	Read/Write
x'04080':x'04FFF'	Reserved	Reserved	
(Implementation-Dependent Area. Refer to the specific implementation documentation for a detailed description of these registers)			
x'05000':x'0FFFF'	PV2_ImplRegs	Privilege 2 implementation-dependent registers	
Reserved Area			
x'10000':x'1FFFF'	Reserved	Reserved	

1. An implementation should support reading of these registers for diagnostic purposes

Table A-5 contains a list of all CBEA-compliant PPE registers, which require Privilege 1 access. For information on each of the registers, see the relevant page.

Table A-5. PPE Privilege 1 Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
Replacement-Management Table Area (RMT) (Implementation Dependent Area: Refer to the specific implementation documentation.)			
x'300':x'2FF'	Reserved	Reserved	
x'300'	L2_RMT_Index	RMT Index Register (see page 233). Index of the replacement-management tables.	Read/Write
x'310'	L2_RMT_Data	RMT Data Register (see page 234). Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation dependent.	Read/Write
x'318':x'7FF'	Reserved	Reserved	
Implementation-Dependent Area (Refer to the specific implementation documentation for a detailed description of these registers)			
x'800':x'FFF'	PUPV_ImplRegs	Implementation dependent PPE privileged-state registers.	

A.4 Internal Interrupt Controller Memory Map

The IIC has an interrupt control block for each physical and logical PPE (that is, a thread in the physical processor). These control blocks are mapped in the real address space, starting at an implementation-dependent offset from the BP_Base. The control block's starting address is defined as the BP_Base | IIC(p) + x’400’ + (t * x’20’); where 'p' is the physical PPE and 't' is the thread number for the corresponding PPE.

(See *Table A-1 Broadband Processor Memory Map* on page 269 for more details.)

Table A-6 shows the registers associated with each control block and their offsets from starting address of the BP_Base.

Table A-6. Internal Interrupt Controller Memory Map

Offset (Hexadecimal)	Register	Description	Access Type
PPU (Thread t) Interrupt Control Block where 0 ≤ t < number of PPU threads -1			
x‘000’	INT_Pending_NonD	Interrupt Pending Port Registers (see page 239) Status and data for pending interrupt	Read Only
x‘008’	INT_Pending_D	Interrupt Pending Port Registers (see page 239), destructive read Status and data for pending interrupt	Read Only
x‘010’	INT_Generation	Interrupt Generation Port Register (see page 243) Port for generation of an interprocessor interrupt (IPI)	Write Only
x‘018’	INT_CPL	Interrupt Current Priority Level Register (see page 244) Only higher-priority interrupts cause an external interrupt	Read/Write

Appendix B. SPU Channel Map

This appendix contains the mapping of all channels defined by the Cell Broadband Engine Architecture in the physical address space. The channel map for a single CBEA-compliant processor is divided into several sections:

- SPU Event Channels
- SPU Signal Notification Channels
- SPU Decrementer Channels
- SPU Multisource Synchronization Channel
- Mask Read Channels
- SPU State Management Channels
- MFC Command Parameter Channels
- MFC Tag Status Channels
- SPU Mailbox Channels

Note: No reserved channels can be used for implementation-dependent functions.

Table B-1. SPU Channel Map (Page 1 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
SPU Event Channels			
x‘0’	SPU_RdEventStat	SPU Read Event Status Channel (see page 136). Read event status (with mask applied).	Read-blocking
x‘1’	SPU_WrEventMask	SPU Write Event Mask Channel (see page 140). Write event-status mask.	Write
x‘2’	SPU_WrEventAck	SPU Write Event Acknowledgment Channel (see page 144). Write end-of-event processing.	Write
SPU Signal Notification Channels			
x‘3’	SPU_RdSigNotify1	SPU Signal Notification 1 Channel (see page 126)	Read-blocking
x‘4’	SPU_RdSigNotify2	SPU Signal Notification 2 Channel (see page 127).	Read-blocking
x‘5’	Channel 5	Reserved	
x‘6’	Channel 6	Reserved	
SPU Decrementer Channels			
x‘7’	SPU_WrDec	SPU Write Decrementer Channel (see page 128).	Write
x‘8’	SPU_RdDec	SPU Read Decrementer Channel (see page 129).	Read
MFC Multisource Synchronization Channels			
x‘9’	MFC_WrMSSyncReq	MFC Write Multisource Synchronization Request Channel (see page 132).	Write-blocking
SPU Reserved Channel			
x‘A’	Channel 10	Reserved	
Mask Read Channels			
x‘B’	SPU_RdEventMask	SPU Read Event Mask Channel (see page 142).	Read
x‘C’	MFC_RdTagMask	MFC Read Tag-Group Query Mask Channel (see page 115).	Read

Table B-1. SPU Channel Map (Page 2 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
SPU State Management Channels			
x‘D’	SPU_RdMachStat	SPU Read Machine Status Channel (see page 130).	Read
x‘E’	SPU_WrSRR0	SPU Write State Save-and-Restore Channel (see page 131).	Write
x‘F’	SPU_RdSRR0	SPU Read State Save-and-Restore Channel (see page 131).	Read
MFC Command Parameter Channels			
x‘10’	MFC_LSA	MFC Local Storage Address Channel (see page 107). Write local storage address command parameter.	Write
x‘11’	MFC_EAH	MFC Effective Address High Channel (see page 109). Write high-order MFC effective-address command parameter.	Write
x‘12’	MFC_EAL	MFC Effective Address Low or List Address Channel (see page 108). Write low-order MFC effective-address command parameter.	Write
x‘13’	MFC_Size	MFC Transfer Size or List Size Channel (see page 106). Write MFC transfer-size command parameter.	Write
x‘14’	MFC_TagID	MFC Command Tag Identification Channel (see page 105). Write TAG identifier command parameter.	Write
x‘15’	MFC_Cmd MFC_ClassID	MFC Command Opcode Channel (see page 103). Write and enqueue MFC command with associated class ID.	Write-blocking
		MFC Class ID Channel (see page 103). Write and enqueue MFC command with associated command opcode.	
MFC Tag Status Channels			
x‘16’	MFC_WrTagMask	MFC Write Tag-Group Query Mask Channel (see page 114). Write TAG mask.	Write
x‘17’	MFC_WrTagUpdate	MFC Write Tag Status Update Request Channel (see page 116). Write request for conditional or unconditional tag-status update.	Write-blocking
x‘18’	MFC_RdTagStat	MFC Read Tag-Group Status Channel (see page 117). Read TAG status (with mask applied).	Read-blocking
x‘19’	MFC_RdListStallStat	MFC Read List Stall-and-Notify Tag Status Channel (see page 118). Read MFC list stall-and-notify status.	Read-blocking
x‘1A’	MFC_WrListStallAck	MFC Write List Stall-and-Notify Tag Acknowledgment Channel (see page 119). Write MFC list stall-and-notify acknowledgment.	Write
x‘1B’	MFC_RdAtomicStat	MFC Read Atomic Command Status Channel (see page 120). Read atomic command status.	Read-blocking
SPU Mailboxes			
x‘1C’	SPU_WrOutMbox	SPU Write Outbound Mailbox Channel (see page 122). Write outbound SPU mailbox contents.	Write-blocking

Table B-1. SPU Channel Map (Page 3 of 3)

Channel Number (Hexadecimal)	Channel Name	Description	Access Type
x‘1D’	SPU_RdInMbox	SPU Read Inbound Mailbox Channel (see page 124). Read inbound SPU mailbox contents.	Read-blocking
x‘1E’	SPU_WrOutIntrMbox	SPU Write Outbound Interrupt Mailbox Channel (see page 123). Write SPU outbound interrupt mailbox contents.	Write-blocking
x’1F’ — x’3F’	Channel 31 — Channel 63	Reserved	

Appendix C. CBEA-Specific PPE Special Purpose Registers

Table C-1 lists the special-purpose registers (SPRs) required by the Cell Broadband Engine Architecture. These registers are accessed using the move-to special-purpose register (**mtspr**) instruction and move from special-purpose register (**mfspr**) Power instruction.

Note: This is not a complete list of SPRs. There can be additional SPRs for specific implementations. Refer to *Book III* of the *Power Architecture* or the specific implementation documentation for a complete list of SPRs.

Table C-1. PPE Special-Purpose Register Map (Page 1 of 2)

Offset	Register	Description	Access Type
Version Register			
x‘3FE’	BP_VR	CBEA-Compliant Processor Version Register (see page 263).	Read Only
Replacement-Management Table Area (RMT) (Implementation-dependent area: Refer to the specific implementation documentation for more information)			
x‘3B6’	PPE_TLB_RMT_Index	RMT Index Register (see page 233). Index of the RMTs.	Read/Write
x‘3B7’	PPE_TLB_RMT_Data	RMT Data Register (see page 234). Doubleword of RMT data pointed to by the RMT Index Register. Entry contents are implementation-dependent.	Read/Write
Instruction-Range SPRs			
x‘3D0’	IRSR0	Instruction Range-Start Register 0 (duplicated per thread).	Read/Write
x‘3D1’	IRMR0	Instruction Range-Mask Register 0 (duplicated per thread).	Read/Write
x‘3D2	ICIDR0	Instruction Class ID Register 0 (duplicated per thread).	Read/Write
x‘3D3	IRSR1	Instruction Range-Start Register 1 (duplicated per thread).	Read/Write
x‘3D4’	IRMR1	Instruction Range-Mask Register 1 (duplicated per thread).	Read/Write
x‘3D5	ICIDR1	Instruction Class ID Register 1 (duplicated per thread).	Read/Write
Data Range SPRs			
x‘3B8’	DRSR0	Data Range-Start Register 0 (duplicated per thread).	Read/Write
x‘3B9’	DRMR0	Data Range-Mask Register 0 (duplicated per thread).	Read/Write
x‘3BA’	DCIDR0	Data Class ID Register 0 (duplicated per thread).	Read/Write
x‘3BB’	DRSR1	Data Range-Start Register 1 (duplicated per thread).	Read/Write
x‘3BC’	DRMR1	Data Range-Mask Register 1 (duplicated per thread).	Read/Write
x‘3BD’	DCIDR1	Data Class ID Register 1 (duplicated per thread).	Read/Write

1. Reading of these registers should be allowed for diagnostic purposes.

Table C-1. PPE Special-Purpose Register Map (Page 2 of 2)

Offset	Register	Description	Access Type
TLB Management SPRs			
x‘3B2’	PPE_TLB_Index_Hint	TLB Index Hint Register (see page 186). Index of best TLB entry to update.	Read Only
x‘3B3’	PPE_TLB_Index	TLB Index Register (see page 187). Index of TLB entry to update with TLB Real Page Number Register and TLB Virtual Page Number Register.[1]	Write Only
x‘3B4’	PPE_TLB_RPN	TLB Real Page Number Register (see page 189). Access to lower portion of TLB entry.	Read/Write
x‘3B5’	PPE_TLB_VPN	TLB Virtual Page Number Register (see page 188). Access to upper portion of TLB entry.	Read/Write

[1] Reading of these registers should be allowed for diagnostic purposes.

Appendix D. Defined Commands

This appendix contains the tables of defined commands from *Section 7 MFC Commands* beginning on page 47. These are the same tables that are shown in that section.

Each of these commands can contain one or more of the parameters listed in *Table D-1 Parameter Mnemonics*.

Defined commands fall into one of three categories:

- Data transfer commands are shown in *Table D-2 Data Transfer or MFC DMA Commands* on page 286
  - Data moved from local storage and placed in main storage (**put** commands)
  - Data moved into local storage from main storage (**get** commands)
- SL1 cache-management commands are shown in *Table D-3 SL1 Storage Control Commands* on page 287.
- Synchronization commands are shown in*Table D-4 Synchronization Commands* on page 287.

The data transfer commands are further divided into sub-categories which define the direction of the data movement (that is, to or from local storage, or **get** and **put**). An application can place the data transfer commands listed in *Table D-2* into the command queue. Unless otherwise noted, these commands can be executed out of order.

Note: Embedded fencing and synchronization commands *must* be used to ensure proper ordering when ordering is required.

Table D-1. Parameter Mnemonics

Parameter	Parameter Name	Register Name	See Note
CL	MFC Class ID	MFC_ClassID	
TG	MFC Command Tag	MFC_Tag	
TS	MFC Transfer Size	MFC_Size	1
LSZ	MFC List Size	MFC_Size	1
LSA	MFC Local Storage Address	MFC_LSA	
EAH	MFC Effective Address High	MFC_EAH	4
EAL	MFC Effective Address Low	MFC_EAL	2
LA	MFC List Local Storage Address	MFC_EAL	2
LTS	List Element Transfer Size		3
LEAL	List Element Effective Address Low		3

1. TS and LSZ share the same register offset. The meaning of the contents depends on the suffix of the MFC opcode.
2. EAL and LA share the same register offset. The meaning of the contents depends on the suffix of the MFC opcode.
3. No associated registers. These parameters are located in local storage and are referenced by the list address (LA) parameter
4. This parameter is optional

Table D-2 shows the Data Transfer, or DMA Commands available in the CBEA.

Table D-2. Data Transfer or MFC DMA Commands (Page 1 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
Put Commands			
put	x‘20’	Proxy/Channel	Moves data from local storage to the effective address.
puts	x‘28	Proxy	Moves data from local storage to the effective address and starts the SPU after the DMA operation completes.
putr	x‘30’	Proxy/Channel	Same as **put** with a PPE L2-cache scarf hint (used to send results to the PPE).
putf	x‘22’	Proxy/Channel	Moves data from local storage to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putb	x‘21’	Proxy/Channel	Moves data from local storage to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putfs	x‘2A’	Proxy	Moves data from local storage to the effective address with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after DMA operation completes.
putbs	x‘29’	Proxy	Moves data from local storage to the effective address with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue) and starts the SPU after DMA operation completes.
putrf	x‘32’	Proxy/Channel	Same as **putf** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrb	x‘31’	Proxy/Channel	Same as **putb** with a PPE L2-cache scarf hint (used to send results to the PPE).
putl	x‘24’	Channel	Moves data from local storage to the effective address using an MFC list.
putrl	x‘34’	Channel	Same as **putl** with a PPE L2-cache scarf hint (used to send results to the PPE).
putlf	x‘26’	Channel	Moves data from local storage to the effective address using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
putlb	x‘25’	Channel	Moves data from local storage to the effective address using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).
putrlf	x‘36’	Channel	Same as **putlf** with a PPE L2-cache scarf hint (used to send results to the PPE).
putrlb	x‘35’	Channel	Same as **putlb** with PPE L2-cache scarf hint (used to send results to PPE).
Get Commands			
get	x‘40’	Proxy/Channel	Moves data from the effective address to local storage.
gets	x‘48’	Proxy	Moves data from the effective address to local storage, and starts the SPU after DMA operation completes.
getf	x‘42’	Proxy/Channel	Moves data from the effective address to local storage with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getb	x‘41’	Proxy/Channel	Moves data from the effective address to local storage with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Table D-2. Data Transfer or MFC DMA Commands (Page 2 of 2)

Mnemonic	Opcode	Support (Proxy/Channel)	Description
getfs	x'4A'	Proxy	Moves data from the effective address to local storage with fence (this command is locally ordered with respect to all previously issued commands within the same tag group), and starts the SPU after DMA operation completes.
getbs	x'49'	Proxy	Moves data from the effective address to local storage with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue), and starts the SPU after DMA operation completes.
getl	x'44'	Channel	Moves data from the effective address to local storage using an MFC list.
getlf	x'46'	Channel	Moves data from the effective address to local storage using an MFC list with fence (this command is locally ordered with respect to all previously issued commands within the same tag group and command queue).
getlb	x'45'	Channel	Moves data from the effective address to local storage using an MFC list with barrier (this command and all subsequent commands with the same tag ID as this command are locally ordered with respect to all previously issued commands within the same tag group and command queue).

Table D-3 lists the available SL1 storage control commands available in the CBEA.

Table D-3. SL1 Storage Control Commands

Mnemonic	Opcode	Support	Description
sdcrt	x'80'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA gets).[1]
sdcrtst	x'81'	Proxy/Channel	Brings a range of effective addresses into the SL1 (performance hint for DMA puts).[1]
sdcrz	x'89'	Proxy/Channel	Writes zeros to the contents of a range of effective addresses.
sdcrst	x'8D'	Proxy/Channel	Stores the modified contents of a range of effective addresses.
sdcrf	x'8F'	Proxy/Channel	Stores the modified contents of a range of effective addresses and invalidates the block.

[1] These commands are treated as no-operations in implementations without an SL1.

Table D-4 lists the synchronization commands available in the CBEA.

Table D-4. Synchronization Commands

Command	Opcode	Support	Description
getllar	x'D0'	Channel	Get lock line and create a reservation (executed immediately).
putllc	x'B4'	Channel	Put lock line conditional on a reservation (executed immediately).
putlluc	x'B0'	Channel	Put lock line unconditional (executed immediately).
putqlluc	x'B8'	Channel	Put lock line unconditional (queued form).
sndsig	x'A0'	Proxy/Channel	Update signal-notification registers in an I/O device or another SPU.
sndsigf	x'A2'	Proxy/Channel	Update signal-notification registers in an I/O device or another SPU with fence.
sndsigb	x'A1'	Proxy/Channel	Update signal-notification registers in an I/O device or another SPU with barrier.

Table D-4. Synchronization Commands

Command	Opcode	Support	Description
barrier	x'C0'	Proxy/Channel	Barrier type ordering. Ensures ordering of all preceding, non-immediate DMA commands with respect to all commands following the barrier command within the same command queue. The barrier command has no effect on the immediate DMA commands: **getllar**, **putllc**, and **putlluc**.
mfceieio	x'C8'	Proxy/Channel	Controls the ordering of get commands with respect to put commands, and of get commands with respect to get commands accessing storage that is caching inhibited and guarded. Also controls the ordering of **put** commands with respect to **put** commands accessing storage that is memory coherence required and not caching inhibited
mfcsync	x'CC'	Proxy/Channel	Controls the ordering of DMA **put** and **get** operations within the specified tag group with respect to other processing units and mechanisms in the system

Appendix E. Extensions to the PowerPC Architecture

The following requests for changes (RFCs) to the PowerPC Architecture are currently being considered for the CBEA. More information will be provided on these extensions in the future.

- Mediated interrupt
- Multiple concurrent large pages
- Add TL (TLB load control) bit to logical-partitioning control register (LPCR) for software versus hardware load of TLB
- Change **tlbie<*l*>** of large page to always invalidate ERAT
- Hypervisor SPRs no longer readable when HV equals 0

Appendix F. Examples of Access Ordering

This appendix contains examples of access ordering as discussed in *Section 10* beginning on page 157.

The following examples demonstrate the order of accesses originated by an SPU, MFC, or PPE. In these examples MFC, SPU and PPE are each shown executing a sequence of instructions or commands. No order between sequences is implied merely by the relative line position of any instruction or command in one sequence compared to the line position of an instruction or command in another sequence. Unless otherwise stated, main storage locations in these examples have main storage attributes of coherence required and not caching inhibited. Local storage alias in these examples have main storage attributes of caching inhibited when accessed from the main storage domain.

See *Section 8.8.1 MFC Multisource Synchronization Register* beginning on page 96 for additional examples for cumulative ordering across the local storage and main storage domains.

Example 1: putf to main storage

SPU (0)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

MFC (0)

1. **put** TG = '1': copy local storage location B to the main storage location D.
2. **putf** TG = '1': copy local storage location A to the main storage location C.

PPE (0)

1. Load main storage location C.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Load main storage location D.

If main storage location D is written with the new value of local storage location B, then main storage location C is written with the new value of local storage location A. After PPE (0) loads the new value of location C, it can load either the old or new value of location D. The local storage accesses related to the **put** and **putf** accesses are ordered, but the **put** and **putf** main storage accesses in the main domain are not ordered.

Example 2: putf to remote local storage

SPU (0)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

MFC (0)

1. **put** TG = '1': copy local storage location B of SPU (0) to local storage location D of SPU (1).
2. **putf** TG = '1': copy local storage location A of SPU (0) to local storage location C of SPU (1).

SPU (1)

1. Load local storage location C.
2. **dsync**
3. Load local storage location D.

If local storage location D is written with the new value of local storage location B, then local storage location C is written with the new value of local storage location A. When SPU (1) loads the new value of local storage location C, it can load either the old or the new value of local storage location D. The local accesses to local storage of SPU (0) related to the **put** and **putf** accesses are ordered, but the **put** and **putf** accesses to the main storage domain from local storage of SPU (1) are not ordered.

Example 3: getf versus PPE stores

MFC (0)

1. **get** TG = ‘1’: read main storage location A.
2. **getf** TG = ‘1’: read main storage location B.

PPE (0)

1. Store to main storage location B.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to main storage location A.

If MFC (0) gets the new value of location A, it can get either the old or new value of B. The **get** and **getf** accesses are locally ordered with respect to local storage, but not with respect to the main storage domain.

Example 4: getf versus remote SPU stores

MFC (0)

1. **get** TG = ‘1’: read local storage location A of SPU (1).
2. **getf** TG = ‘1’: read local storage location B of SPU (1).

SPU (1)

1. Store to local storage location B.
2. **dsync**
3. Store to local storage location A.

If MFC (0) gets the new value of location A, it can get either the old or new value of B. The **get** and **getf** accesses are locally ordered with respect to local storage, but not with respect to local storage of another SPU.

Example 5: getf versus local SPU loads (This example is only used to demonstrate storage access ordering. It is not a recommended programming method. Typically, software would use the Tag Group Completion facility to get predictable new values for the SPU Loads.)

MFC (0)

1. **get** TG = ‘1’: copy main storage location C to local storage location A of SPU (0).
2. **getf** TG = ‘1’: copy main storage location D to local storage location B of SPU (0).

SPU (0)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

When SPU (0) loads the new value of location B, it must also load the new value of A.

Example 6: Tag-independent **barrier** command versus local SPU loads

MFC (0)

1. **get** TG = ‘1’: copy main storage location X to local storage location A.
2. **barrier** TG = ‘2’.
3. **get** TG = ‘3’: copy main storage location Y to local storage location B.

SPU (0)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (0) loads the new value of location B, it must load the new value of A.

Example 7: Tag-group status completion and **put** versus PPE loads

This sequence assumes the Proxy Tag-Group Query Mask Register (see page 84) is already set so tag group 1 is part of a query.

SPU (0)

1. Enqueue a **put** TG = ‘1’ to write main storage location A.
2. Request an immediate tag-status update and wait for Tag Group Completion.
   For information on the Proxy Tag-Group Completion Facility (see page 82), and for information on the MFC Tag-Group Status Channels (see page 111).
3. Enqueue a **put** TG = ‘1’ to write main storage location B.

PPE (0)

1. Load from main storage location B.
2. Use any serialization mechanism that orders these loads.
3. Load from main storage location A.

If PPE (0) loads the new value of location B, PPE(0) can load either the old or the new value of A. The DMA accesses are ordered with respect to local storage of SPU (0) by the Tag Group Status completion status, but not ordered with respect to the main storage domain.

Example 8: Tag-group status completion and **put** versus local SPU loads

This sequence assumes the Proxy Tag-Group Query Mask Register (see page 84) is already set so that tag group 1 is part of a query.

SPU (0)

1. Enqueue a **put** TG = ‘1’ to write main storage location A.
2. Request an immediate tag-status update and wait for Tag Group Completion.
   For information on the Proxy Tag-Group Completion Facility (see page 82), and for information on the MFC Tag-Group Status Channels.
3. Enqueue a **put** TG = ‘1’ to write main storage location B.

SPU (1)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If SPU (1) loads the new value of location B, it can load either the old or the new value of A. The DMA accesses for each **put** command are locally ordered with respect to local storage of SPU (0) by the Tag-Group Status completion status, but not ordered with respect to the local storage of another SPU.

Example 9: mfcsync and **put** versus PPE loads

MFC (0)

1. **put** TG = ‘1’ to write main storage location A
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to write main storage location B

PPE (0)

1. Load main storage location B
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III.*)
3. Load main storage location A.

If **PPE** (0) loads the new value of location B, it must also load the new value of A.

Example 10: mfcsync and **put** versus remote SPU local storage loads

MFC (0)

1. **put** TG = ‘1’ to write local storage location A of SPU (1).
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to write local storage location B of SPU (1)

SPU (1)

1. Load from local storage location B.
2. **dsync**
3. Load from local storage location A.

If **SPU (1)** loads the new value of location B, it must also load the new value of A.

Example 11: mfcsync and **put** versus local SPU local storage stores (This example is only used to demonstrate storage access ordering. If it is only necessary to ensure local storage access order and not main storage access order, then a tag specific fence or barrier is sufficient.)

MFC (0)

1. **put** TG = ‘1’ to read from local storage location A.
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to read from local storage location B.

SPU (0)

1. Store to local storage location B.
2. **dsync**
3. Store to local storage location A.

If **MFC (0)** reads the new value of location A, it must also read the new value of B.

Example 12: mfcsync and **get**

MFC (0)

1. **get** TG = ‘1’ to read from main storage location A.
2. **mfcsync** TG = ‘1’
3. **get** TG = ‘1’ to read from main storage location B.

PPE (0)

1. Store to main storage location B.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to main storage location A.

If **MFC (0)** gets the new value of location A, it must also get the new value of B.

Example 13: Cumulative Ordering

MFC (0)

1. **put** to write main storage location A.

PPE (0)

1. Load from main storage location A.
2. **sync**
3. Store to main storage location B.

MFC (1)

1. **get** TG = ‘1’ to read main storage location B.
2. **mfcsync** TG = ‘1’
3. **get** TG = ‘1’ to read main storage location A.

If PPE (0) loads the new value of location A and MFC (1) gets the new value of B, MFC (1) must also get the new value of A.

Example 14: Cumulative Ordering

MFC (0)

1. **put** TG = ‘1’ to write main storage location A.
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to write main storage location B.

PPE (0)

1. Loop loading from main storage location B until the new value is loaded.
2. Store to main storage location C.

MFC (1)

1. **get** TG = ‘1’ to read main storage location C.
2. **mfcsync** TG = ‘1’
3. **put** TG = ‘1’ to read main storage location A.

If MFC (1) gets the new value of C, MFC (1) must also get the new value of A.

Example 15: SPU stores versus PPE loads

SPU (1)

1. Store to local storage location A.
2. **dsync**
3. Store to local storage location B.

PPE (0)

1. Load local storage location B.
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III.*)
3. Load local storage location A.

If PPE (0) loads the new value of location B, it must also load the new value of location A.

Example 16: SPU loads versus PPE stores

SPU (1)

1. Load from local storage location B
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to local storage location B

If SPU (1) loads the new value of location B, it must also load the new value of A.

Example 17: SPU loads versus PPE stores to local storage and SPU Signal Notification 1 Register (see page 94). In this example, if more than two units are involved in the set of storage accesses that must be completed before the store to the SPU Signal Notification 1 Register, then MFC Multisource Synchronization Register (see page 96) must be used. For more information, see *Section 8.8.1 MFC Multisource Synchronization Register* beginning on page 96.

SPU (1)

1. Read from SPU Signal Notification 1 Register
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to SPU Signal Notification 1 Register

If SPU (1) loads the new value of the SPU Signal Notification 1 Register, it must also load the new value of A.

Example 18: SPU loads versus PPE stores to local storage and SPU Inbound Mailbox Register (see page 92)

SPU (1)

1. Read from SPU Inbound Mailbox Register (see page 92).
2. **dsync**
3. Load from local storage location A.

PPE (0)

1. Store to local storage location A.
2. Use any serialization mechanism that causes these stores to be performed in order (see *PowerPC Architecture, Books I-III*).
3. Store to SPU Inbound Mailbox Register.

If SPU (1) loads the new value of the SPU Inbound Mailbox Register, it must also load the new value of A.

Example 19: mfceieio and **put** versus PPE loads

MFC (0)

1. **put** TG = ‘1’ to write main storage location A
2. **mfceieio** TG = ‘1’
3. **put** TG = ‘1’ to write main storage location B

PPE (0)

1. Load main storage location B
2. Use any serialization mechanism that causes these loads to be performed in order (see *PowerPC Architecture, Books I-III.*)
3. Load main storage location A.

If PPE (0) loads the new value of location B, it must also load the new value of A.

Example 20: mfceieio and **get**

Main storage locations A and B have the storage attributes of caching inhibited and guarded.

MFC (0)

1. **get** TG = ‘1’ to read from main storage location A.
2. **mfceieio** TG = ‘1’
3. **get** TG = ‘1’ to read from main storage location B.

PPE (0)

1. Store to main storage location B.
2. **eieio**
3. Store to main storage location A.

If MFC (0) gets the new value of location A, it must also get the new value of B.

Glossary

BIC	bus interface controller
BIU	bus interface unit
BMT	bandwidth management table
caching inhibited	A page of storage is considered caching inhibited when the “I” bit has a value of ‘1’ in the page table. Data located in caching inhibited pages cannot be cached at any memory hierarchy that is not visible to all processors and devices in the system. Stores must update the memory hierarchy to a level that is visible to all processors and devices in the system.
CBEA	Cell Broadband Engine Architecture
CL	DMA class ID
CSA	context-save area
CSRA	context save and restore area
DAR	data-address register
dcbf	data cache block flush
dcbst	data cache block store
dcbt	data-cache block touch x-form
dcbtst	data cache block touch for store
dcbz	data cache block zero
DMA	direct memory access
DR	data relocate
DSI	data-storage interrupt
DSISR	data-storage interrupt-status register
EA	effective address
EAH	DMA effective address high
EAL	DMA effective address low
ECC	error correction code
EIC	external-interrupt controller
eieio	enforce in-order execution of I/O transaction
ERAT	effective-address-to-real-address translation
ESID	effective segment ID

fence	Barrier type fence. Waits for completion of all preceding DMA commands before starting any commands queued after any command with the **fence** option. (Does not apply to the **getllar**, **putllc** or **putlluc** commands, which are immediately executed.)
FLIH	first-level interrupt handler
fres	floating reciprocal estimate single A-form
frsqte	floating reciprocal square-root estimate A-form
getllar	get lock line and reserve command
GPR	general-purpose register
HID	Hardware implementation dependent
hypervisor	A facility that provides and manages multiple virtual machines
IABR	instruction address breakpoint register
ICBI	instruction cache block invalidate
IIC	internal-interrupt controller
INT	interrupt packet
interrupt packet	Used to signal an interrupt, typically to a processor or to another interruptible device.
IOC	I/O controller
IOIF	I/O interface
IPI	interprocessor interrupt
IR	instruction relocate
ISRC	interrupt source
JTAG	Joint Test Action Group
L1	A memory cache built in the CPU chip.
L2	A memory cache that is external to the CPU chip.
LA	DMA list address
ld	load doubleword instruction
ldarx	load doubleword and reserve x-form instruction
LEAL	list element effective address low
livelock	An endless loop in program execution
lmw	load multiple word

LPAR	logical partitioning
LPID	logical-partition identity
LRU	least recently used
LS	local storage
LSA	local storage address
LSA	DMA local storage size
LSb	least-significant bit
LSCSA	local storage context-save area
lswi	load string word immediate
lswx	load string word indexed
LTS	list element transfer size
lwarx	load word and reserve x-form
MFC	memory flow controller
mfceieio	MFC enforce in-order execution of I/O command
mfcsync	MFC synchronize command
mfspr	move from special-purpose register
MIC	memory interface controller
MMIO	memory-mapped I/O
MMU	memory-management unit
MSb	most significant bit
mtmsr	move to machine state register instruction
mtspr	move to special-purpose register instruction
PG	processing-unit group
PME	Privileged Mode Environment
PPE	PowerPC processor element
PPU	PowerPC processor unit
privileged software	Software that has access to the Privileged Modes of the architecture.
PTE	page-table entry
putllc	put lock line conditional on a reservation
putlluc	put lock line unconditional

PVR	processor version register
QoS	quality of service
RA	real address
rchcnt	read channel count instruction
rdch	read from channel instruction
real address	The address of a byte in real storage or a byte on an I/O device
RMT	replacement-management table
RPN	real-page number
scarfing	The direct transfer of data to the PPE L2 cache
SCEI	Sony Computer Entertainment Incorporated
sdcrf	SL1 data-cache range flush
sdcrst	SL1 data-cache range store
sdcrt	SL1 data-cache range touch
sdcrtst	SL1 data-cache range touch for store
sdcrz	SL1 data-cache range set to zero
SDR	storage descriptor register
SG	SPU group
SIMD	single instruction, multiple data
SL1	A first-level cache for DMA transfers between local storage and main storage
SLB	segment lookaside buffer
slbia	SLB invalidate all instruction
slbie	SLB invalidate entry instruction
slbmfee	SLB move-from entry ESID X-form instruction
slbmfev	SLB move-from entry VSID X-form instruction
slbmte	SLB move-to entry X-form instruction
sndsig	send signal command
sndsigb	Update signal-notification registers in an I/O device or another SPU with barrier
sndsigf	Update signal-notification registers in an I/O device or another SPU with fence
SPE	Synergistic processor element, which consists of a synergistic processor unit (SPU) and a memory flow controller (MFC)

SPR	special-purpose register
SPU	synergistic processor unit
SRR1	save and restore register 1
stdcx	store doubleword conditional indexed instruction
stmw	store multiple word instruction
storage model	For definition of storage models, see *Section 3* on page 33.
stswi	store string word immediate instruction
stswx	store string word indexed x-form instruction
stwcx	store word conditional x-form instruction
sync	synchronize instruction
tag group	A set of commands with the same tag identifier is defined as a tag group.
TG	tag parameter
TLB	translation lookaside buffer
tlbie	translation lookaside buffer invalidate entry
TS	DMA transfer size
UME	User Mode Environment
VA	virtual address
VPN	virtual-page number
VS	virtual storage
VSID	virtual segment ID
WIMG bits	Four bits in the page table, also called a page-table entry, which control the processor's accesses to cache and main storage. “W” stands for write through, “I” for cache inhibit, “M” for memory coherence, and “G” for guarded storage.
wrch	write to channel

Index

Revision Log

Revision Date	Version	Contents of Modification
August 8, 2005	1.0	Public Release of Document.